

COMP-9: Configuration — Component Specification

1. Purpose & Intent

COMP-9 exists to solve a fundamental adoption problem: architecture analysis tools historically demanded upfront, manual, project-specific configuration before producing any value. This created a chicken-and-egg barrier — users had to understand their architecture well enough to describe it in config before the tool could help them understand their architecture.

The core philosophy is “point at any repo and get a valid config.” COMP-9 implements a self-bootstrapping configuration system where `get_config()` always succeeds. If a `.architecture-model.yaml` file exists, it’s loaded. If not, the component scans the filesystem, infers project layout, discovers functional blocks from directory structure, and synthesizes a complete `ProjectConfig` — all transparently.

This makes COMP-9 the **spine of the entire system**. Every other component reads configuration through it. It translates the physical reality of a repository (directories, files, naming conventions) into the logical abstractions (layers, functional blocks, metrics) that the rest of the pipeline operates on.

2. Goals & Measures of Effectiveness

Goal	Optimization Target	Measure of Effectiveness	Acceptable Threshold
Universal applicability	Any Python repo works without config	% of repos where <code>get_config()</code> returns a usable <code>ProjectConfig</code> without manual intervention	>95% for standard layouts (src-layout, flat-layout)
Discovery accuracy	Heuristic scanning claims real source files	<code>DiscoveryReport.claim_rate</code> — ratio of <code>files_claimed</code> to <code>files_total</code>	>90% file claim rate
Layer detection correctness	Inferred layers match actual architecture	Layers discovered vs. layers a human would identify	>80% precision for Python web projects
Zero friction entry	Single function call, no setup ceremony	<code>get_config()</code> never raises on a valid directory	100% — must always return a <code>ProjectConfig</code>
Round-trip fidelity	<code>to_dict()</code> → YAML → <code>from_dict()</code> preserves all fields	Field-level equality after round-trip	Lossless for all serializable fields
Sub-block granularity	Auto-discovered blocks reflect real package structure	Sub-blocks match Python subpackages with <code>.py</code> files	1:1 correspondence with code-containing subdirectories

Key trade-off: Convention over configuration. The heuristic tables (`_LAYER_HEURISTICS`, `_METRIC_HEURISTICS`) encode assumptions about Python web project structure. These will produce false positives on non-standard layouts (e.g., a directory named `models/` that contains ML models, not ORM models). This is explicitly acceptable because the alternative — requiring manual config — has a worse failure mode: users produce no architectural model at all. False positives can be corrected by writing a `.architecture-model.yaml`; missing config cannot be corrected without the tool.

3. Architecture Role & Position

COMP-9 sits in the **infrastructure layer** — it has no domain logic of its own but provides the foundational data structures every domain component depends on. It is the most depended-upon component in the system.

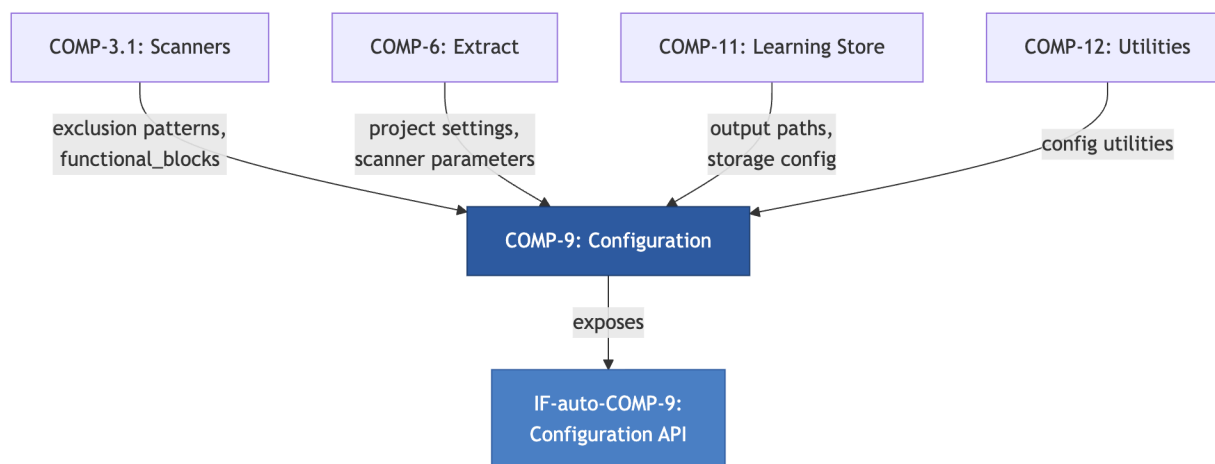


Figure 1: Diagram

COMP-9 has **zero outbound component dependencies** (except `utils.discovery.EXCLUDED_DIRS`), making it a leaf in the dependency graph — exactly where infrastructure belongs.

4. API Surface

`get_config(root: Path) -> ProjectConfig`

The recommended entry point. Always returns a valid config.

```
def get_config(root: Path) -> ProjectConfig
```

Parameter	Type	Description
<code>root</code>	<code>Path</code>	Project root directory

Behavior: Checks for `.architecture-model.yaml`. If present, loads it via `load_config()`. If the loaded YAML lacks `functional_blocks` (e.g., it's a model file, not a project config), falls back to `discover_config()`. For blocks loaded from YAML that lack `sub_blocks`, auto-discovers them from the filesystem. If no config file exists, runs full discovery.

```
load_config(root: Path) -> ProjectConfig
```

Loads config strictly from file. Raises `FileNotFoundError` if missing.

```
def load_config(root: Path) -> ProjectConfig
```

Raises: `FileNotFoundError` with a message suggesting `architecture-model init` or `get_config()`.

```
discover_config(root: Path) -> tuple[ProjectConfig, DiscoveryReport]
```

Full auto-discovery. Returns both the synthesized config and an observability report.

```
def discover_config(root: Path) -> tuple[ProjectConfig, DiscoveryReport]
```

The `DiscoveryReport` contains layout type detected, counts of discovered entities, candidate evaluations, and the file claim rate.

```
write_config(config: ProjectConfig, root: Path | None = None) -> Path
```

Serializes a `ProjectConfig` to `.architecture-model.yaml` with an auto-generation header comment.

```
load_profile(name_or_path: str) -> DomainProfile
```

Loads a domain profile by builtin name ("software", "controls", "mechanical", "electrical") or filesystem path.

```
from architecture_model.profiles.schema import load_profile
profile = load_profile("controls")
```

Resolution Flow

5. Data Model

ProjectConfig

The root configuration object. All other components consume this.

Field	Type	Default	Purpose
name	str	""	Project name, used in output path templates
system	str	""	System identifier
output	OutputConfig	OutputConfig()	Output path templates

Field	Type	Default	Purpose
layers	list[LayerConfig[]]		Architecture tiers
functional_blocks	list[FunctionalBlockConfig]		Capability decomposition
metrics	list[MetricConfig[]]		Countable project metrics
root	Path	Path(".")	Filesystem root (not serialized)

Computed properties:

Property	Return Type	Purpose
layer_dir_map	dict[str, list[str]]	Layer ID → directories (for merger)
source_block_dir_map	dict[str, str]	Directory/file prefix → block ID (for merger heuristics)
source_block_dict	dict[str, dict]	Legacy dict format for backward compatibility
metrics_paths	dict[str, Path]	Metric label → resolved absolute path

OutputConfig / ResolvedOutputConfig

Field	Default	Example resolved
model	"output/{project}/architecture-model.yaml"	realty/output/myapp/architecture-model.yaml
manifest	"output/{project}/reality-manifest.json"	realty/output/myapp/reality-manifest.json
artifacts	"output/{project}/artifacts/stage2"	realty/output/myapp/artifacts/stage2

LayerConfig

Field	Type	Purpose
id	str	e.g. "web-layer", "data-layer"
dirs	list[str]	Directories belonging to this layer
description	str	Optional human description

FunctionalBlockConfig / SubBlockConfig

Recursive capability decomposition mirroring package nesting.

Field	Type	Purpose
id	str	e.g. "S1", "S1.A", "S1.A.1"
name	str	Human-readable, from dir name or docstring

Field	Type	Purpose
<code>dirs</code>	<code>list[str]</code>	Relative directories
<code>files</code>	<code>list[str]</code>	Relative <code>.py</code> file paths (excluding <code>__init__.py</code>)
<code>description /</code> <code>description_source</code>	<code>str</code>	From <code>__init__.py</code> docstring or auto-generated
<code>sub_blocks</code>	<code>list[SubBlockConfig]</code>	Nested children (recursive)

ID scheme: top-level blocks use `S{n}`, first-level subs use `S{n}.{A-Z}`, deeper levels use `S{n}.{letter}.{digit}`.

MetricConfig

Field	Type	Default	Purpose
<code>label</code>	<code>str</code>	—	e.g. "router", "model"
<code>path</code>	<code>str</code>	—	Directory to count in
<code>pattern</code>	<code>str</code>	<code>"*.py"</code>	Glob pattern
<code>exclude</code>	<code>list[str]</code>	<code>[]</code>	Filenames to skip
<code>recursive</code>	<code>bool</code>	<code>False</code>	Use <code>rglob</code> vs <code>glob</code>

DiscoveryReport

Field	Type	Purpose
<code>layout_detected</code>	<code>str</code>	"src-layout", "flat-layout", "lib-layout", "fallback", "unknown"
<code>blocks_discovered</code>	<code>int</code>	Count of F-blocks found
<code>sub_blocks_discovered</code>	<code>int</code>	Count of sub-blocks found
<code>claim_rate</code>	<code>float (property)</code>	<code>files_claimed / files_total</code>
<code>candidates</code>	<code>list[DiscoveryCandidate]</code>	Audit trail of evaluated paths

6. Auto-Discovery Engine

Layer Discovery

`_discover_layers()` checks the `_LAYER_HEURISTICS` table:

Layer ID	Candidate Directories
<code>web-layer</code>	<code>app/routers</code> , <code>app/views</code> , <code>app/api</code> , <code>src/api</code> , <code>api/</code>
<code>services-layer</code>	<code>app/services</code> , <code>src/services</code> , <code>services/</code>
<code>data-layer</code>	<code>app/models</code> , <code>src/models</code> , <code>models/</code> , <code>alembic</code>
<code>pipeline-layer</code>	<code>scripts</code> , <code>pipeline</code> , <code>src/pipeline</code> , <code>jobs/</code>
<code>scheduling-layer</code>	<code>app/tasks</code> , <code>tasks/</code> , <code>celery/</code>

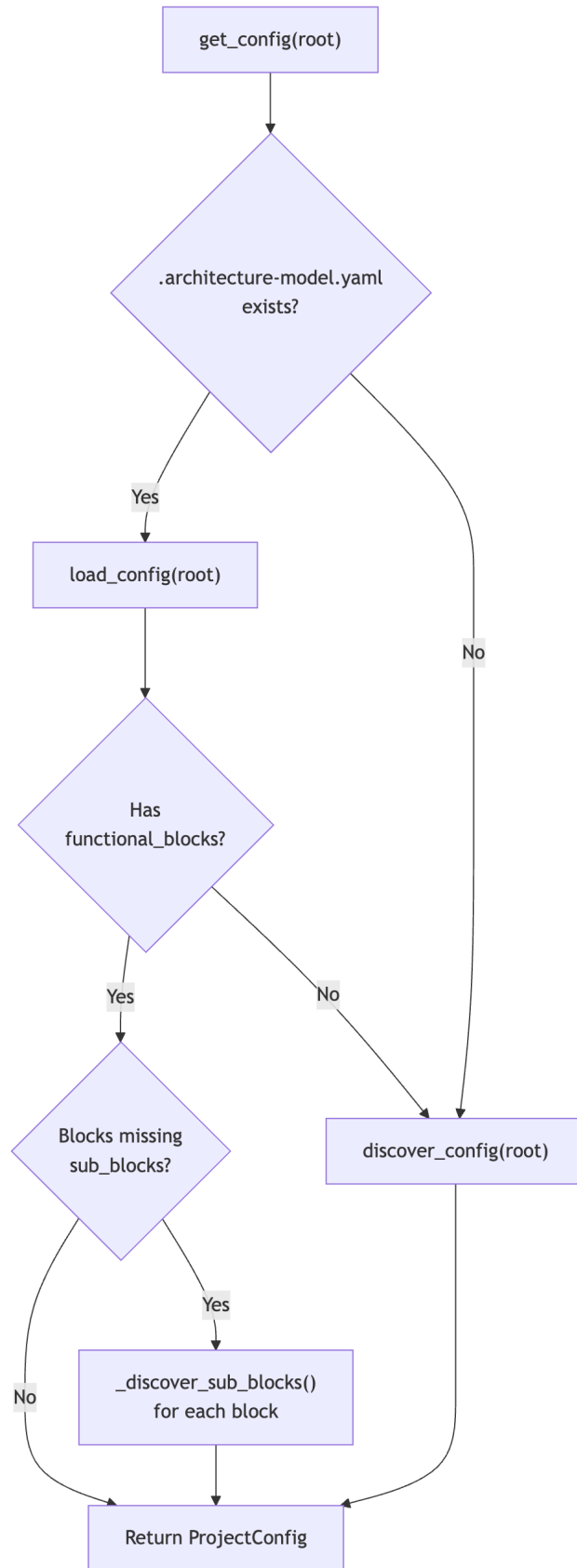


Figure 2: Diagram

Metric Discovery

`_discover_metrics()` uses `_METRIC_HEURISTICS` — first matching path per label wins.

Functional Block Discovery

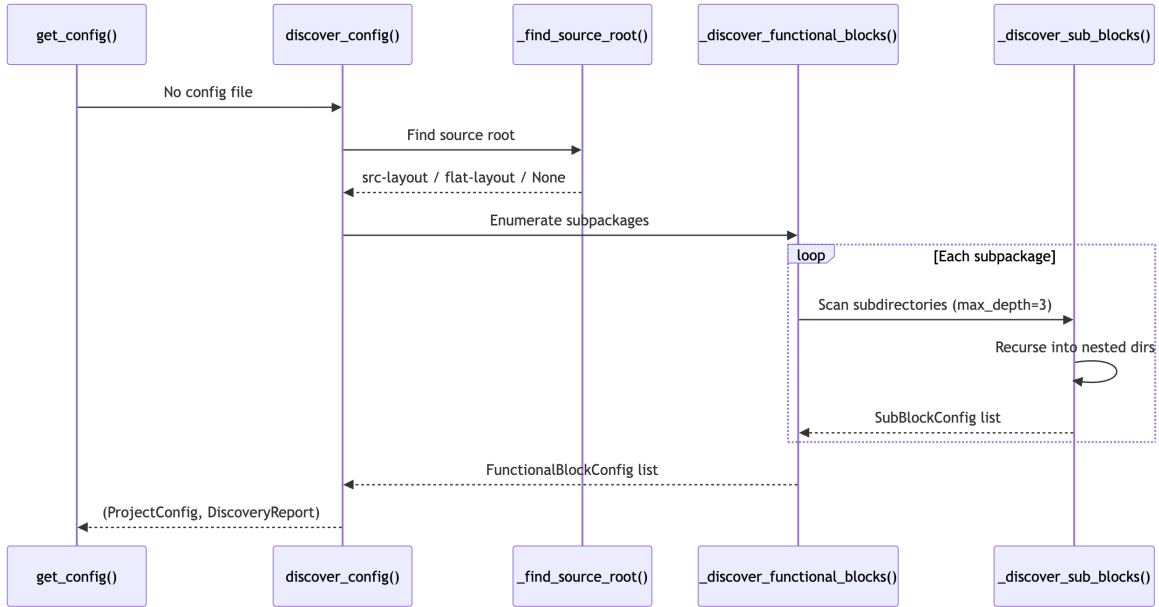


Figure 3: Diagram

Source root detection (`_find_source_root`): Tries `src/<pkg>/` → `flat <pkg>/` → `lib/<pkg>/` in order. Requires `__init__.py` and at least one code-containing subdirectory.

Sub-block discovery (`_discover_sub_blocks`): Recurses up to `max_depth=3`. Only creates sub-blocks for directories containing `.py` files. Extracts names from `__init__.py` docstrings via `ast.parse`.

Fallback: `_source_blocks_from_top_level_dirs()` when no package structure is detected.

7. Domain Profiles

The profile system enables COMP-9 to serve projects beyond software — controls engineering, mechanical, electrical domains.

Built-in Profiles

Name	File	Domain
software	software.yaml	Default software architecture
controls	controls.yaml	Control systems (sensors, actuators)
mechanical	mechanical.yaml	Mechanical assemblies
electrical	electrical.yaml	Electrical systems

DomainProfile Structure

Dataclass	Purpose
<code>EnumExtension</code>	Adds values to schema enums (e.g., new <code>ComponentKind</code> values)
<code>EntityExtension</code>	Adds properties to entity types
<code>ConditionalRule</code>	<code>when condition</code> → <code>require</code> fields with <code>message</code>

```
profile = load_profile("controls")
extra_kinds = profile.get_extended_values("ComponentKind")
```

8. Design Decisions & Rationale

Decision	Rationale
Typed dataclasses over raw dicts	Type safety, IDE autocomplete, construction-time validation. <code>ProjectConfig.from_dict()</code> catches schema mismatches early.
Auto-discovery over mandatory init	Reduces adoption cost from “write 50-line YAML” to “run the tool.” Discovery can always be overridden by writing config.
Convention-based layer detection	80/20 rule — the 5 heuristic patterns cover most Python web projects. Non-matching projects get <code>_derive_layers_from_blocks()</code> fallback.
Recursive sub-blocks with <code>max_depth=3</code>	Mirrors real package nesting. Depth cap prevents runaway scanning on deep vendor trees.
YAML config format	Already the format for <code>.architecture-model.yaml</code> model output; no new dependency. Human-readable and editable.
<code>get_config()</code> as single entry point	Eliminates decision fatigue for consumers. They never need to know if config was file-based or discovered.
<code>DiscoveryReport</code> observability	Debugging heuristic-based systems requires audit trails. <code>candidates</code> list shows what was evaluated and why it was accepted/rejected.
<code>{project}</code> template in output paths	Supports multi-project analysis from a single root without path collisions.

9. Consumers & Integration Patterns

Consumer	What They Use	How
COMP-3.1 (Scanners)	<code>functional_blocks[].dirs</code> , <code>EXCLUDED_DIRS</code>	Scope file discovery to known source directories; skip excluded paths
COMP-6 (Extract)	<code>get_config()</code> → full <code>ProjectConfig</code>	Reads project settings, passes config to scanners
COMP-11 (Learning Store)	<code>resolved_output()</code> , <code>root</code>	Determines where to read/write pipeline state
COMP-12 (Utilities)	Various config fields	General-purpose config access
CLI / Pipeline	<code>get_config(root)</code> as first call	Every command starts by resolving config; downstream stages receive <code>ProjectConfig</code>

Integration pattern: All consumers call `get_config()` (never `discover_config()` directly). This ensures consistent behavior regardless of whether a config file exists.

10. Constraints & Limitations

- **YAML-only** — no TOML, JSON, or programmatic configuration support
- **Python-web-centric heuristics** — `_LAYER_HEURISTICS` targets FastAPI/Django/Flask layouts; TypeScript and Kotlin projects get minimal layer detection (fallback only)
- **No hot-reload** — config is loaded once per invocation; file changes require re-running
- **No config schema validation** — relies on dataclass construction and `from_dict()` defaults rather than JSON Schema or Pydantic validation
- **Directory-based discovery only** — sub-blocks are inferred from filesystem structure, not import graphs
- **Monorepo limitations** — `_find_source_root()` returns the first matching package; multi-package monorepos may only discover one
- **Heuristic false positives** — a `models/` directory containing ML models will be tagged as data-layer
- **max_depth=3 for sub-blocks** — deeply nested packages beyond 3 levels are not represented

11. Requirements Traceability with Rationale

ID	Requirement	Rationale	MoE
CFG-R1	<code>get_config()</code> shall return a valid <code>ProjectConfig</code> for any directory without raising exceptions (except OS-level errors).	The zero-friction goal requires that consumers never need to handle “no config” cases. Every directory has <i>some</i> valid configuration, even if it’s minimal.	100% success rate on valid directories
CFG-R2	<code>load_config()</code> shall raise <code>FileNotFoundError</code> with actionable guidance when <code>.architecture-model.yaml</code> is absent.	Distinguishes “explicit load” from “best-effort load.” The error message suggests <code>init</code> or <code>get_config()</code> so users aren’t stuck.	Error message contains both alternatives
CFG-R3	Auto-discovery shall detect <code>src-layout</code> , <code>flat-layout</code> , and <code>lib-layout</code> Python projects.	These three layouts cover >95% of Python packages per PyPA packaging guide.	<code>DiscoveryReport.layout_d</code> is not “unknown” for standard projects
CFG-R4	<code>ProjectConfig.from_dict()</code> and <code>to_dict()</code> shall round-trip losslessly for all serializable fields.	Config files must survive read-modify-write cycles (e.g., <code>init</code> then manual edit then re-read).	Field equality after <code>from_dict(config.to_dict</code>
CFG-R5	Sub-block discovery shall not recurse beyond <code>max_depth=3</code> levels.	Prevents unbounded filesystem traversal on large vendor directories or deeply nested generated code. 3 levels captures package → subpackage → module grouping.	Verified by <code>_discover_sub_blocks</code> depth parameter
CFG-R6	<code>write_config()</code> shall include auto-generation header comments.	Users must know a file was generated (not hand-written) so they understand they can edit it. Prevents confusion about file provenance.	Output file starts with <code># Architecture Model Standard</code> comment block
CFG-R7	Discovery shall produce a <code>DiscoveryReport</code> with candidate evaluation audit trail.	Heuristic systems are opaque by default. The report enables debugging why a directory was or wasn’t claimed as a block.	<code>report.candidates</code> list is non-empty after discovery
CFG-R8	Domain profiles shall be loadable by builtin name or filesystem path.	Builtin profiles serve common domains; filesystem paths enable custom/proprietary profiles without modifying the package.	<code>load_profile("controls")</code> and <code>load_profile("/path/to/c</code> both succeed

ID	Requirement	Rationale	MoE
CFG-R9	<code>ProjectConfig</code> shall expose <code>source_block_dir_map</code> and <code>layer_dir_map</code> as computed properties.	Consumers (merger, scanners) need fast lookups from directory paths to logical entities. Computing on access avoids stale caches while keeping the data model clean.	Properties return correct mappings for all configured blocks/layers
CFG-R10	The configuration file name shall be <code>.architecture-model.yaml</code> (dot-prefixed).	Dot-prefixed files are hidden by default on Unix, keeping project roots clean. The name is self-documenting and unlikely to collide with other tools.	<code>CONFIG_FILENAME</code> constant equals <code>".architecture-model.yam</code>

Model Completeness: D (35%) Some sections may be empty due to missing model entities. - 1/1 components have no behavioral specification - No requirements defined - No actors defined → conops stakeholder section empty - No constraints defined → operations manual empty Run the extraction pipeline or manually add behaviors/interfaces/constraints.

Artifact Traceability Map: architecture-model-standard / Configuration

1. Entity Inventory

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

2. Artifact Dependency Graph

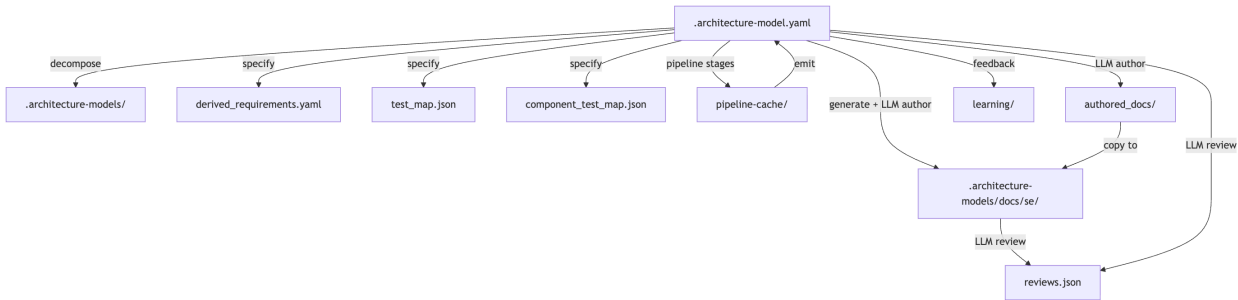


Figure 4: Diagram

3. Entity-to-Artifact Traceability Matrix

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

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Component1	1							
Specs								
ConOps		—					—	
Functional		—	—					
Analysis								
Interface	1			1				
Specifi- cation								
Logical	1			1				—
Archi- tecture								
Maintenance	1							
Manual								
Operations	1							
Manual								
Requirements		—			—	—		
Analysis								
Risk					—			
Assess- ment								
Use			—				—	
Cases								
Verification			—			—		
& Vali- dation								

4. Relationship Distribution

Relationship Type	Count	Connects
depends-on	5	Unknown → Component, Unknown → Unknown
exposes	1	Component → Interface

5. Traceability Gaps

- **Capabilities** — 0 entities; leaves gaps in: ConOps, Functional Analysis, Requirements Analysis
- **Behaviors** — 0 entities; leaves gaps in: Use Cases, Functional Analysis, Verification & Validation, Behavior Flows
- **Constraints** — 0 entities; leaves gaps in: Requirements Analysis, Risk Assessment
- **Requirements** — 0 entities; leaves gaps in: Requirements Analysis, Verification & Validation
- **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
- **realizes** relationship type missing — weakens cross-entity traceability