

OpenCode Architecture Extension

Software Engineering Documentation

Generated by opencode-arch-docs

2026-07-08

Contents

1	System Overview: OpenCode Architecture Extension	6
1.1	Project	6
1.2	Architecture	6
1.3	Key Components	7
1.3.1	CLI Commands	7
1.3.2	MCP Server	7
1.3.3	Prompt Templates	7
1.3.4	Learning Loop	7
1.3.5	OpenCode Runner	7
1.3.6	Telemetry Store	8
1.4	Component Interactions	8
1.5	Constraints	8
1.6	Capabilities	8
2	Functional Architecture — opencode-arch	10
2.1	1. Purpose	10
2.2	2. Functional Block Decomposition	10
2.2.1	F1 — Core Workflows (Primary Value)	10
2.2.2	F3 — Intelligence (Learning)	11
2.2.3	F4 — Agent Interface (MCP)	11
2.2.4	F7 — Observability	11
2.3	3. Actor Model	12
2.3.1	3.1 Actors	12
2.3.2	3.2 Actor-Capability Access Matrix	12
2.4	4. Use-Case Mapping	12
2.4.1	4.1 Behavior Catalog	12
2.4.2	4.2 Use-Case to Capability Mapping	13
2.4.3	4.3 Detailed Use-Case Flows	13
2.4.3.1	BEH-EXTRACT: Extract Architecture	13
2.4.3.2	BEH-REGEN: Regen-Loop Iteration	13
2.4.3.3	BEH-DOCS: Generate Documentation	14
2.5	5. Traceability Matrix	14
2.6	6. Functional Interaction Diagram	15
2.7	7. Functional Constraints	16
2.8	8. Capability Realization Summary	17
2.9	9. Cross-Functional Dependencies	17

2.10	10. Navigational Traceability Diagram	18
2.11	11. Focused View Example: CAP-DOCS	18
2.11.1	How to Use for Task Assignment	19
3	Capability Map — opencode-arch	20
3.1	Overview	20
3.2	Capabilities	20
3.2.1	CAP-EXTRACT — Architecture Extraction	20
3.2.2	CAP-GENERATE — Code Generation	20
3.2.3	CAP-DOCS — SE Document Generation	21
3.2.4	CAP-REGEN — Subsystem Regen-Loop	21
3.2.5	CAP-MCP — MCP Tool Serving	21
3.2.6	CAP-LEARN — Learning Loop	22
3.2.7	CAP-TELEMETRY — Telemetry & Metrics	22
3.3	Realization Summary	22
3.4	Component Dependency Graph	23
3.5	Constraints	23
4	Component Reference Catalog	24
4.1	Overview	24
4.2	Components	24
4.2.1	COMP-CLI: CLI Commands	24
4.2.2	COMP-MCP: MCP Server	24
4.2.3	COMP-RUNNER: OpenCode Runner	25
4.2.4	COMP-LEARNING: Learning Loop	25
4.2.5	COMP-TELEMETRY: Telemetry Store	26
4.2.6	COMP-PROMPTS: Prompt Templates	26
4.3	Component Dependencies	26
4.3.1	Direct Dependencies	26
4.3.2	Capability Realizations	27
4.3.3	Constraints Applied	27
4.4	Dependency Summary	27
5	Layer Architecture — opencode-arch	28
5.1	Overview	28
5.2	Layers	28
5.2.1	Layer 1: CLI Layer (LAYER-CLI)	28
5.2.2	Layer 1: MCP Server Layer (LAYER-MCP)	28
5.2.3	Layer 2: Learning Layer (LAYER-LEARNING)	29
5.2.4	Layer 3: Runner Layer (LAYER-RUNNER)	29
5.2.5	Layer 3: Telemetry Layer (LAYER-TELEMETRY)	29
5.3	Layer Interaction Map	29
5.3.1	Dependency Graph	29
5.3.2	Dependency Summary	30
5.3.3	External Interfaces	31
5.4	Capability Realization	31
5.5	Constraints	31
5.6	Layer Ordering Compliance	31

6	Behavior Flows	33
6.1	Overview	33
6.2	BEH-EXTRACT: Extract Architecture	33
6.2.1	Preconditions	33
6.2.2	Steps	33
6.2.3	Postconditions	34
6.2.4	Sequence Diagram	34
6.3	BEH-REGEN: Regen-Loop Iteration	34
6.3.1	Preconditions	34
6.3.2	Steps	35
6.3.3	Postconditions	35
6.3.4	Sequence Diagram	35
6.4	BEH-DOCS: Generate Documentation	36
6.4.1	Preconditions	36
6.4.2	Steps	36
6.4.3	Postconditions	36
6.4.4	Sequence Diagram	36
6.5	BEH-MCP-SCAN: MCP Scan Tool	37
6.5.1	Preconditions	37
6.5.2	Steps	37
6.5.3	Postconditions	37
6.6	Pattern Summary	37
7	Dependency Graph — opcode-arch	39
7.1	Overview	39
7.2	Direct Dependencies	39
7.2.1	Grouped by Source Component	39
7.2.1.1	COMP-CLI (CLI Commands)	39
7.2.1.2	COMP-LEARNING (Learning Loop)	39
7.2.1.3	COMP-MCP (MCP Server)	39
7.3	Capability Realization	40
7.4	Constraints	40
7.5	Dependency Analysis	40
7.5.1	Coupling Metrics	40
7.5.2	Highly-Coupled Components	40
7.5.3	Circular Dependency Analysis	40
7.5.4	Observations	41
7.5.5	Suggested Improvements	41
8	Integration Guide — opcode-arch	42
8.1	Overview	42
8.2	Available Interfaces	42
8.2.1	IF-MCP — MCP Tool Protocol	42
8.2.2	IF-CLI — CLI Interface	42
8.2.3	IF-RUNNER — Runner Backend Protocol	43
8.2.4	IF-TELEMETRY — Telemetry Store Interface	43
8.2.5	IF-ARCH-MODEL — Architecture Model API	43
8.3	Integration Patterns	43

8.3.1	Integrating with the MCP Server (COMP-MCP)	43
8.3.2	Integrating with CLI Commands (COMP-CLI)	44
8.3.3	Integrating with the OpenCode Runner (COMP-RUNNER)	44
8.3.4	Integrating with the Learning Loop (COMP-LEARNING)	44
8.3.5	Integrating with Telemetry (COMP-TELEMETRY)	45
8.3.6	Integrating with Prompt Templates (COMP-PROMPTS)	45
8.4	Component Dependency Graph	45
8.5	Security Constraints & Authentication	46
8.5.1	CON-PERMISSIONS — Cross-Directory Access	46
8.5.2	CON-NO-HALLUCINATION — No Hallucination Constraint	46
8.6	Performance Constraints	46
8.6.1	CON-TOKENS — Token Budget Constraint	46
8.6.2	CON-TIMEOUT — Runner Timeout	46
8.7	Summary of Integration Points	47
9	API Reference — opcode-arch	48
9.1	Overview	48
9.2	IF-CLI: CLI Interface	48
9.2.1	Entry Point	48
9.2.2	Endpoints	48
9.2.2.1	1. extract	48
9.2.2.2	2. generate	49
9.2.2.3	3. bench	49
9.2.2.4	4. metrics	49
9.2.2.5	5. report	49
9.2.2.6	6. regen-loop	50
9.3	IF-MCP: MCP Tool Protocol	50
9.3.1	Server Configuration	50
9.3.2	Endpoints	50
9.3.2.1	1. architect_scan	50
9.3.2.2	2. architect_slice	51
9.3.2.3	3. architect_validate	52
9.3.2.4	4. architect_extract	52
9.3.2.5	5. architect_generate	53
9.4	IF-RUNNER: Runner Backend Protocol	54
9.4.1	Protocol Definition	54
9.4.2	Endpoint	54
9.4.2.1	run	54
9.4.3	Default Implementation: OpenCodeRunner	54
9.5	IF-TELEMETRY: Telemetry Store Interface	55
9.5.1	Database Location	55
9.5.2	Endpoints	55
9.5.2.1	1. record	55
9.5.2.2	2. query	56
9.5.2.3	3. averages	56
9.5.3	Async Wrapper	56
9.6	IF-ARCH-MODEL: Architecture Model API	57
9.6.1	Dependency	57

9.6.2	Endpoints	57
9.6.2.1	1. generate_manifest()	57
9.6.2.2	2. format_model_context()	57
9.6.2.3	3. validate_model()	57
9.6.2.4	4. Slicer API	57
9.7	Data Contracts	58
9.7.1	Architecture Model YAML Schema	58
9.7.2	MCP Request/Response Contract	59
9.7.3	RunResult Contract	59
9.7.4	Telemetry Record Contract	59
10	Constraint Register — opcode-arch	60
10.1	Overview	60
10.2	Constraint Definitions	60
10.2.1	CON-TOKENS — Token Budget Constraint	60
10.2.2	CON-NO-HALLUCINATION — No Hallucination Constraint	61
10.2.3	CON-TIMEOUT — Runner Timeout	61
10.2.4	CON-PERMISSIONS — Cross-Directory Access	61
10.3	Constraint Allocation	62
10.3.1	Allocation Diagram	62
10.3.2	Dependency Context	63
10.4	Enforcement Mechanisms	63
11	Deployment View — opcode-arch	65
11.1	Overview	65
11.2	Deployment Units	65
11.2.1	Unit 1: CLI Application	65
11.2.2	Unit 2: MCP Server	66
11.2.3	Unit 3: OpenCode Runner	66
11.2.4	Unit 4: Learning Loop	67
11.2.5	Unit 5: Telemetry Store	67
11.3	Deployment Topology	67
11.4	Operational Constraints	68
11.4.1	Performance	68
11.4.2	Reliability	68
11.4.3	Security	69
11.5	Failure Modes	69
11.6	Prerequisites	70
12	Metrics Dashboard	71
12.1	Code Metrics	71
12.2	Quality Assessment	71
12.2.1	Overall Health	71
12.2.2	Observations	71
12.2.3	Recommendations	71

Chapter 1

System Overview: OpenCode Architecture Extension

1.1 Project

Project: opencode-arch Schema Version: 1.4 Generated: 2026-07-08

The OpenCode Architecture Extension (opencode-arch) is an MCP server and CLI toolset that provides context compression tools for LLM agents working with software codebases. It enables architecture-driven development by compressing repository structure into minimal token representations, allowing frontier models to reason about codebases with reduced token expenditure.

1.2 Architecture

The system is organized into five layers, ordered by proximity to the user:

Layer	Technology	Purpose
CLI Layer	argparse, asyncio	User-facing command interface
MCP Server Layer	FastMCP, mcp-protocol	Agent-facing tool interface
Learning Layer	dataclasses, regex	Pattern classification and prompt adaptation
Runner Layer	subprocess, Protocol	External process invocation
Telemetry Layer	SQLite	Metrics persistence and querying

The CLI and MCP layers serve as the two primary entry points — one for human developers, one for LLM agents. Both delegate downward to the Runner, Learning, and Telemetry layers for execution, intelligence, and observability respectively.

1.3 Key Components

1.3.1 CLI Commands

- Type: Module
- Layer: CLI Layer
- Technology: argparse
- Role: Parses command-line arguments and orchestrates the extraction, generation, regen-loop, and documentation workflows. Displays results to the user.
- Key files: `src/opencode_arch/cli/main.py`, `src/opencode_arch/cli/extract.py`, `src/opencode_arch/cli/generate.py`, `src/opencode_arch/cli/bench.py`, `src/opencode_arch/cli/regen_loop.py`, `src/opencode_arch/cli/docs.py`

1.3.2 MCP Server

- Type: Service
- Layer: MCP Server Layer
- Technology: FastMCP
- Role: Exposes architecture tools via the MCP protocol. Compresses context within token budgets and bridges between the LLM agent and the architecture-model-standard library.
- Key files: `src/opencode_arch/mcp/server.py`, `src/opencode_arch/mcp/tools/scan.py`, `src/opencode_arch/mcp/tools/slice.py`, `src/opencode_arch/mcp/tools/validate.py`, `src/opencode_arch/mcp/tools/extract.py`

1.3.3 Prompt Templates

- Type: Library
- Layer: CLI Layer
- Technology: Python strings
- Role: Defines structured LLM prompt templates for extraction and regeneration workflows.
- Key files: `src/opencode_arch/prompts/regen.py`, `src/opencode_arch/prompts/extract.py`

1.3.4 Learning Loop

- Type: Module
- Layer: Learning Layer
- Technology: dataclasses, regex
- Role: Classifies test failure patterns, adapts prompts based on observed patterns, generates report cards with grades, extracts and stores lessons, and detects documentation drift.
- Key files: `src/opencode_arch/learning/classifier.py`, `src/opencode_arch/learning/adapters.py`, `src/opencode_arch/learning/assessor.py`, `src/opencode_arch/learning/lessons.py`

1.3.5 OpenCode Runner

- Type: Library
- Layer: Runner Layer
- Technology: subprocess, Protocol

- Role: Invokes `opencode` run as a subprocess, handles timeouts and errors, and returns structured `RunResult` objects.
- Key files: `src/opencode_arch/runner/base.py`, `src/opencode_arch/runner/opencode.py`

1.3.6 Telemetry Store

- Type: Data Store
- Layer: Telemetry Layer
- Technology: SQLite
- Role: Persists tool invocation metrics, stores regeneration outcomes and learning data, and provides a query interface for metrics display.
- Key files: `src/opencode_arch/telemetry/store.py`, `src/opencode_arch/telemetry/recorder.py`

1.4 Component Interactions

The system's components interact through well-defined dependency relationships:

CLI as orchestrator: The CLI Commands component serves as the primary orchestrator, depending on four other components: - It invokes the OpenCode Runner to delegate reasoning tasks to the LLM agent via subprocess. - It reads from and writes to the Telemetry Store to record metrics and display results. - It leverages the Learning Loop to classify failures, adapt prompts, and improve over iterations. - It uses Prompt Templates to construct structured prompts for extraction and regeneration workflows.

MCP as agent interface: The MCP Server consumes the Architecture Model API (an external interface provided by the `architecture-model-standard` library) to offer compressed context and validation to LLM agents.

Learning feedback loop: The Learning Loop depends on the Telemetry Store to read historical outcomes and persist lessons learned, enabling prompt adaptation based on accumulated experience.

1.5 Constraints

The system operates under three documented constraints:

Constraint	Applies To	Description
Token Budget	MCP Server	Context compression must stay within the specified token budget
No Hallucination	CLI Commands	Outputs must be grounded in actual repository content
Timeout	OpenCode Runner	Subprocess invocations must respect timeout limits

1.6 Capabilities

The system realizes the following capabilities:

- Architecture Extraction (CLI Commands) — Extract architecture models from repositories
- Code Generation (CLI Commands) — Generate code with test verification
- Documentation (CLI Commands) — Generate and validate documentation
- Regeneration Loop (CLI Commands) — Iterative improvement of generated artifacts
- MCP Tool Exposure (MCP Server) — Provide tools to LLM agents via protocol
- Learning (Learning Loop) — Classify patterns and adapt behavior
- Telemetry (Telemetry Store) — Record and query operational metrics

Chapter 2

Functional Architecture — opencode-arch

2.1 1. Purpose

This document describes the functional decomposition of the OpenCode Architecture Extension system. It maps actors to use cases, capabilities to functional blocks, and traces requirements through to their realizing components — providing a complete functional view of the system.

2.2 2. Functional Block Decomposition

The system is organized into functional blocks (F-blocks), each representing a cohesive area of functionality:

F-Block	Name	Capabilities	Priority
F1	Core Workflows	Architecture Extraction, Code Generation, SE Doc Generation, Subsystem Regen-Loop	High
F3	Intelligence	Learning Loop (pattern classification, prompt adaptation, report cards)	Medium
F4	Agent Interface	MCP Tool Serving (scan, slice, validate, extract, generate)	High
F5	Prompt Engineering	Prompt Templates (extraction, regeneration)	High
F6	Execution	Runner Backend (subprocess invocation, timeout handling)	High
F7	Observability	Telemetry & Metrics (invocation recording, outcome tracking)	Medium

2.2.1 F1 — Core Workflows (Primary Value)

The largest functional block, containing the four main capabilities that deliver user-visible value:

F1: Core Workflows

- └─ CAP-EXTRACT: Architecture Extraction
 - └─ Scan repository AST
 - └─ Produce validated architecture model
 - └─ Achieve score $\geq 80/100$
- └─ CAP-GENERATE: Code Generation
 - └─ Regenerate code from model
 - └─ Run tests to validate
 - └─ Iterate until pass rate met
- └─ CAP-DOCS: SE Document Generation
 - └─ Select artifacts based on model richness
 - └─ Assemble context from model and manifest
 - └─ Generate grounded documentation
- └─ CAP-REGEN: Subsystem Regen-Loop
 - └─ Decompose system into subsystems
 - └─ Iterate per-subsystem until convergence
 - └─ Support blind mode (model-only context)

2.2.2 F3 — Intelligence (Learning)

Closed-loop learning that improves generation quality over successive runs:

F3: Intelligence

- └─ CAP-LEARN: Learning Loop
 - └─ Pattern Classifier (7 types, 14 regex rules)
 - └─ Adaptive Prompt Optimizer (4 heuristic rules)
 - └─ Report Card Assessor (A-F grading)
 - └─ Lesson Extractor (deduplication, storage)

2.2.3 F4 — Agent Interface (MCP)

Token-arbitrage bridge between LLM agents and the architecture model:

F4: Agent Interface

- └─ CAP-MCP: MCP Tool Serving
 - └─ architect_scan → reality manifest
 - └─ architect_slice → compressed context
 - └─ architect_validate → model quality score
 - └─ architect_extract → model persistence
 - └─ architect_generate → test verification

2.2.4 F7 — Observability

Persistent metrics for tracking system evolution:

F7: Observability

- └─ CAP-TELEMETRY: Telemetry & Metrics
 - └─ Tool invocation recording
 - └─ Regen outcome logging

└─ Learning curve data

2.3 3. Actor Model

2.3.1 3.1 Actors

Actor	Type	Interface	Primary Goals
Developer (ACT-DEV)	Human	CLI (argparse)	Extract architecture, generate docs, validate code
LLM Agent (ACT-AGENT)	System	MCP (stdio)	Consume compressed context, produce models, regenerate code
OpenCode Runner (ACT-OPENCODE)	External Service	Subprocess	Execute LLM prompts on behalf of CLI

2.3.2 3.2 Actor-Capability Access Matrix

Capability	Developer	LLM Agent	Runner
Architecture Extraction	Direct	-	Invoked
Code Generation	Direct	-	Invoked
SE Document Generation	Direct	-	Invoked
Subsystem Regen-Loop	Direct	-	Invoked
Learning Loop	Direct	-	-
MCP Tool Serving	-	Direct	-
Telemetry & Metrics	Direct (read)	-	-

Access patterns: - Developer triggers workflows via CLI, which delegate to the Runner for LLM inference - LLM Agent consumes tools via MCP protocol — never invokes CLI directly - Runner is a passive execution backend — invoked by CLI, has no autonomous behavior

2.4 4. Use-Case Mapping

2.4.1 4.1 Behavior Catalog

ID	Use Case	Actor	Trigger	Pattern
BEH-EXTRACT	Extract Architecture	Developer	opencode-arch extract	Sequential
BEH-REGEN	Regen-Loop Iteration	Developer	opencode-arch regen-loop	Pipeline
BEH-DOCS	Generate Documentation	Developer	opencode-arch docs generate	Sequential

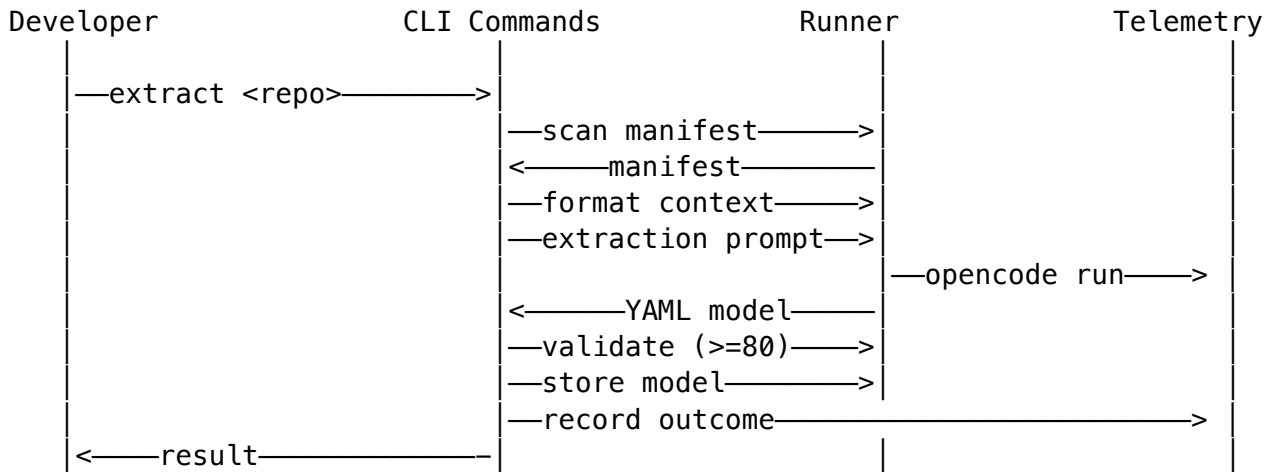
ID	Use Case	Actor	Trigger	Pattern
BEH-MCP-SCAN	MCP Scan Tool	LLM Agent	architect_scan call	Sequential

2.4.2 4.2 Use-Case to Capability Mapping

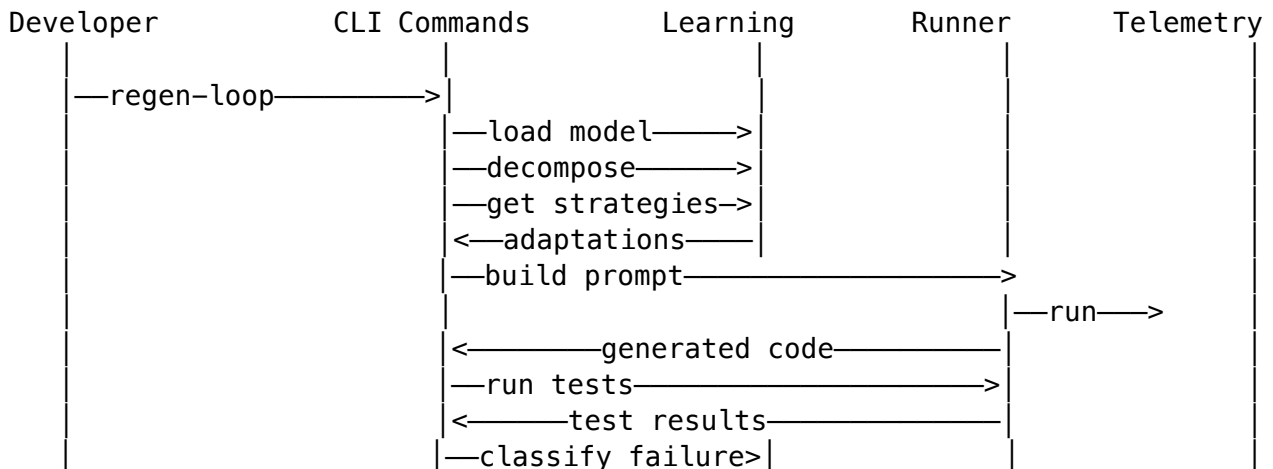
Use Case	Primary Capability	Supporting Capabilities
Extract Architecture	CAP-EXTRACT	CAP-TELEMETRY
Regen-Loop Iteration	CAP-REGEN	CAP-LEARN, CAP-TELEMETRY
Generate Documentation	CAP-DOCS	CAP-TELEMETRY
MCP Scan Tool	CAP-MCP	-

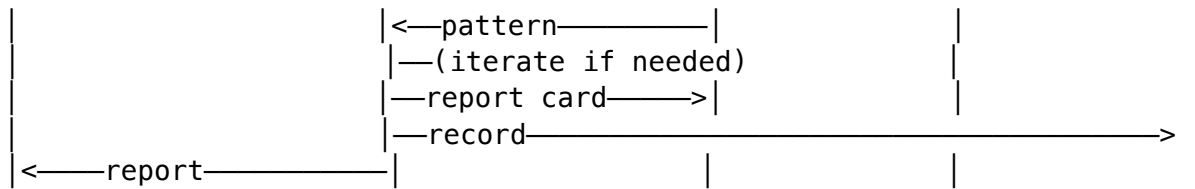
2.4.3 4.3 Detailed Use-Case Flows

2.4.3.1 BEH-EXTRACT: Extract Architecture

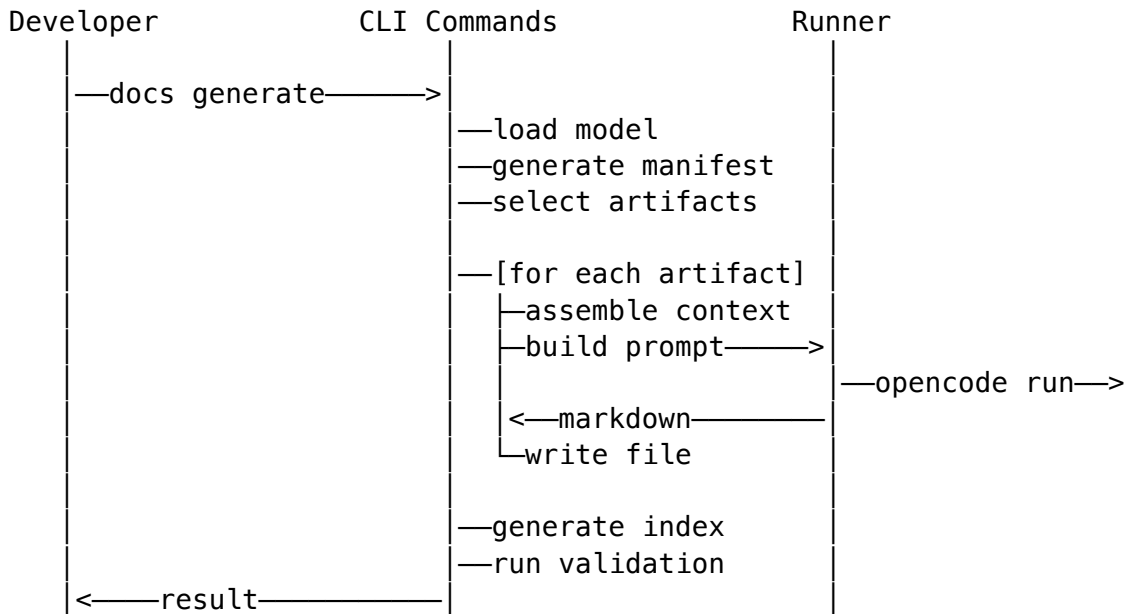


2.4.3.2 BEH-REGEN: Regen-Loop Iteration





2.4.3.3 BEH-DOCS: Generate Documentation



2.5 5. Traceability Matrix

Full requirements traceability from actor goals through to implementing components:

Actor Goal	Behavior	Capability	Component	Interface	Constraint
Extract architecture from repositories	BEH-EXTRACT	CAP-EXTRACT	COMP-CLI, COMP-RUNNER	IF-CLI, IF-RUNNER	CON-NO-HALLUCINATION
Generate SE documentation	BEH-DOCS	CAP-DOCS	COMP-CLI, COMP-RUNNER	IF-CLI, IF-RUNNER	CON-NO-HALLUCINATION, CON-TIMEOUT
Validate code against models	BEH-REGEN	CAP-REGEN	COMP-CLI, COMP-RUNNER, COMP-LEARNING	IF-CLI, IF-RUNNER	CON-TIMEOUT

Actor Goal	Behavior	Capability	Component	Interface	Constraint
Consume compressed context	BEH-MCP-SCAN	CAP-MCP	COMP-MCP	IF-MCP	CON-TOKENS
Produce architecture models	BEH-EXTRACT	CAP-EXTRACT	COMP-MCP	IF-MCP	CON-NO-HALLUCINATION
Regenerate code from models	BEH-REGEN	CAP-REGEN, CAP-GENERATE	COMP-CLI, COMP-RUNNER	IF-CLI, IF-RUNNER	CON-TIMEOUT

2.6 6. Functional Interaction Diagram

```

@startuml Functional Architecture
!theme plain
skinparam componentStyle rectangle

actor "Developer" as dev
actor "LLM Agent" as agent
rectangle "OpenCode Runner" as runner

package "F1: Core Workflows" {
    [Architecture Extraction] as extract
    [Code Generation] as generate
    [SE Doc Generation] as docs
    [Subsystem Regen-Loop] as regen
}

package "F3: Intelligence" {
    [Learning Loop] as learn
}

package "F4: Agent Interface" {
    [MCP Tool Serving] as mcp
}

package "F5: Prompt Engineering" {
    [Prompt Templates] as prompts
}

package "F6: Execution" {
    [Runner Backend] as runnerComp
}

package "F7: Observability" {
    [Telemetry & Metrics] as telemetry

```

```

}

dev --> extract : "CLI"
dev --> generate : "CLI"
dev --> docs : "CLI"
dev --> regen : "CLI"
dev --> telemetry : "read metrics"

agent --> mcp : "MCP stdio"

extract --> runnerComp : "invoke"
generate --> runnerComp : "invoke"
docs --> runnerComp : "invoke"
regen --> runnerComp : "invoke"
regen --> learn : "classify/adapt"

extract --> prompts : "use"
generate --> prompts : "use"
regen --> prompts : "use"

extract --> telemetry : "record"
generate --> telemetry : "record"
regen --> telemetry : "record"
learn --> telemetry : "store lessons"

runnerComp --> runner : "subprocess"

mcp ..> extract : "architect_extract"
mcp ..> generate : "architect_generate"
@enduml

```

2.7 7. Functional Constraints

Constraint	Applies To	Metric	Threshold	Rationale
CON-TOKENS	F4 (MCP)	context_tokens	4000 default	LLM context windows are limited; compression enables larger repos
CON-NO-HALLUCINATION	F1 (Core)	grounding_accuracy	100%	All claims must trace to model/manifest data

Constraint	Applies To	Metric	Threshold	Rationale
CON-TIMEOUT	F6 (Execution)	execution_time	600s	Prevent hung subprocesses from blocking pipeline
CON-PERMISSIONS	F6 (Execution)	filesystem_access	SWD-scoped	Runner operates within specified working directory

2.8 8. Capability Realization Summary

CAP-EXTRACT —realizes→ COMP-CLI —depends→ COMP-RUNNER —depends→ ACT-OPENCODE
CAP-GENERATE —realizes→ COMP-CLI —depends→ COMP-RUNNER —depends→ ACT-OPENCODE
CAP-DOCS —realizes→ COMP-CLI —depends→ COMP-RUNNER —depends→ ACT-OPENCODE
CAP-REGEN —realizes→ COMP-CLI —depends→ COMP-RUNNER —depends→ ACT-OPENCODE
COMP-LEARNING —depends→ COMP-TELEMETRY
CAP-LEARN —realizes→ COMP-LEARNING —depends→ COMP-TELEMETRY
CAP-MCP —realizes→ COMP-MCP —consumes→ IF-ARCH-MODEL
CAP-TELEMETRY realizes→ COMP-TELEMETRY

2.9 9. Cross-Functional Dependencies

The system has a clear dependency hierarchy (no cycles):

From (Consumer)	To (Provider)	Nature
F1 (Core Workflows)	F5 (Prompt Engineering)	Prompt construction
F1 (Core Workflows)	F6 (Execution)	LLM invocation
F1 (Core Workflows)	F7 (Observability)	Metrics recording
F1 (Core Workflows)	F3 (Intelligence)	Pattern adaptation
F3 (Intelligence)	F7 (Observability)	Lesson/pattern storage
F4 (Agent Interface)	External: architecture-model-standard	Model APIs

Dependency direction: F1 → {F3, F5, F6, F7}; F3 → F7; F4 → External

No circular dependencies exist. The system maintains strict layered dependency flow.

2.10 10. Navigational Traceability Diagram

The full-model navigation diagram shows all entities and relationships on one page. Use entity IDs to precisely identify work scope.

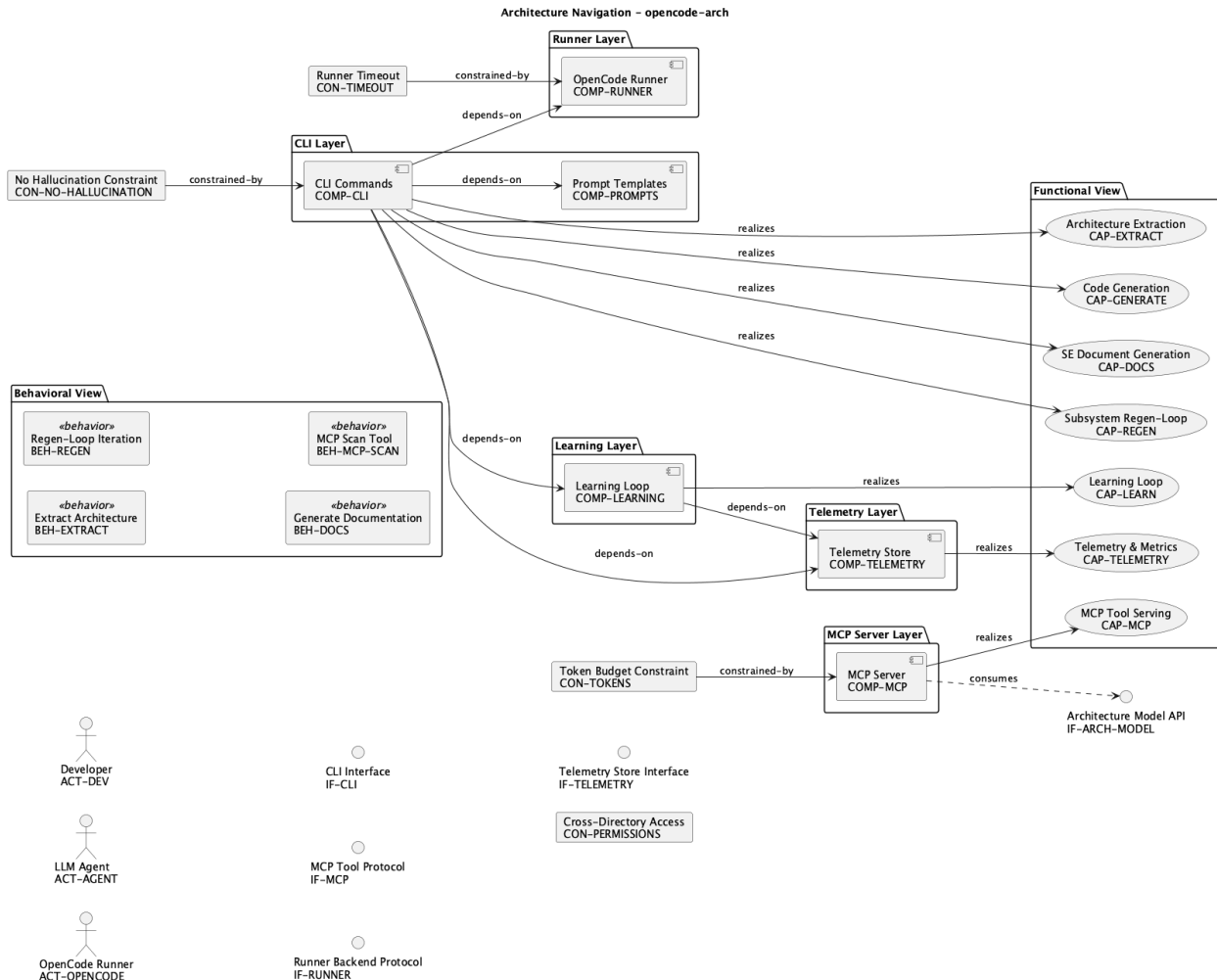


Figure 2.1: Architecture Navigation Diagram

2.11 11. Focused View Example: CAP-DOCS

To isolate work on SE Document Generation, use `generate_focused_diagram(model, "CAP-DOCS", depth=2)`:

Reading this diagram: To work on CAP-DOCS in isolation, the developer needs:

- Implementation scope: COMP-CLI (specifically `src/opencode_arch/cli/docs.py`)
- Direct dependencies: COMP-RUNNER (execution), COMP-PROMPTS (prompt templates)
- Transitive dependencies: COMP-TELEMETRY (metrics), COMP-LEARNING (adaptation)
- Constraints to satisfy: CON-NO-HALLUCINATION (100% grounded in model data)
- Behavioral spec: BEH-DOCS (defines the step sequence)

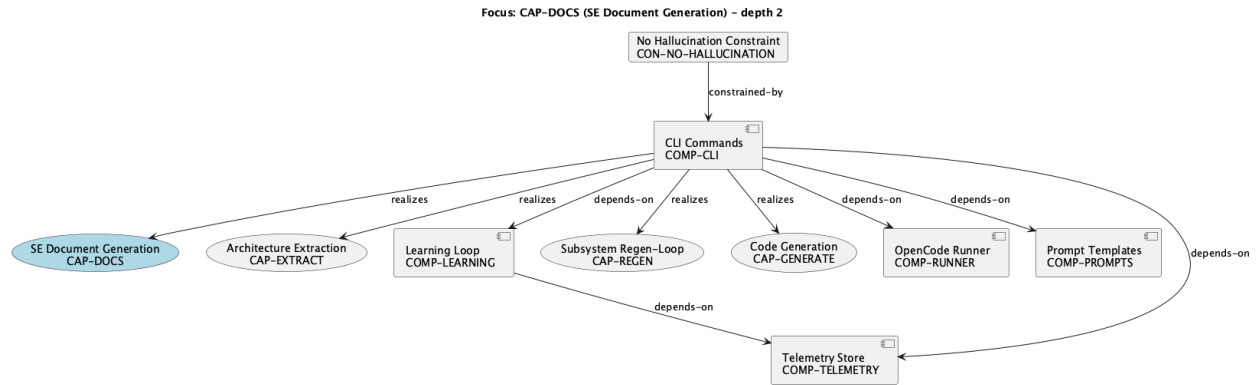


Figure 2.2: Focused View: CAP-DOCS

2.11.1 How to Use for Task Assignment

1. Pick the broken/target capability (e.g., CAP-DOCS)
2. Run `generate_focused_diagram(model, "CAP-DOCS")` to see its neighborhood
3. The “realizes” edge tells you WHICH component to modify
4. The “depends-on” edges tell you what APIs you can call
5. The “constrained-by” edges tell you invariants to maintain
6. Hand the developer: component files + behavior spec + dependency interfaces

Chapter 3

Capability Map — opencode-arch

3.1 Overview

This document maps each system capability to its priority, functional block, realizing components, and supporting dependencies.

3.2 Capabilities

3.2.1 CAP-EXTRACT — Architecture Extraction

Attribute	Value
Priority	High
Functional Block	F1
Realized by	CLI Commands
Dependencies	OpenCode Runner, Telemetry Store, Learning Loop, Prompt Templates
Constraints	CON-NO-HALLUCINATION

Extracts architecture models from target repositories by orchestrating the runner with structured prompts and recording results via telemetry.

3.2.2 CAP-GENERATE — Code Generation

Attribute	Value
Priority	High
Functional Block	F1
Realized by	CLI Commands
Dependencies	OpenCode Runner, Telemetry Store, Learning Loop, Prompt Templates

Attribute	Value
Constraints	CON-NO-HALLUCINATION

Generates code artifacts from architecture models, delegating inference to the runner backend and validating outputs against test suites.

3.2.3 CAP-DOCS — SE Document Generation

Attribute	Value
Priority	High
Functional Block	F1
Realized by	CLI Commands
Dependencies	OpenCode Runner, Telemetry Store, Learning Loop, Prompt Templates
Constraints	CON-NO-HALLUCINATION

Produces software engineering documentation artifacts from architecture context and repository structure.

3.2.4 CAP-REGEN — Subsystem Regen-Loop

Attribute	Value
Priority	High
Functional Block	F1
Realized by	CLI Commands
Dependencies	OpenCode Runner, Telemetry Store, Learning Loop, Prompt Templates
Constraints	CON-NO-HALLUCINATION

Iteratively regenerates subsystem code until quality gates (test pass rate, validation score) are satisfied.

3.2.5 CAP-MCP — MCP Tool Serving

Attribute	Value
Priority	High
Functional Block	F4

Attribute	Value
Realized by	MCP Server
Consumes	Architecture Model API
Constraints	CON-TOKENS

Serves the five architecture tools (scan, slice, validate, extract, generate) over the MCP protocol to LLM agents.

3.2.6 CAP-LEARN — Learning Loop

Attribute	Value
Priority	Medium
Functional Block	F3
Realized by	Learning Loop
Dependencies	Telemetry Store

Analyzes historical telemetry to optimize context budgets and prompt strategies over time.

3.2.7 CAP-TELEMETRY — Telemetry & Metrics

Attribute	Value
Priority	Medium
Functional Block	F7
Realized by	Telemetry Store

Records tool invocations, context token counts, output quality scores, and iteration counts to a persistent store.

3.3 Realization Summary

Capability	Realizing Component	Priority
CAP-EXTRACT	CLI Commands	High
CAP-GENERATE	CLI Commands	High
CAP-DOCS	CLI Commands	High
CAP-REGEN	CLI Commands	High
CAP-MCP	MCP Server	High

Capability	Realizing Component	Priority
CAP-LEARN	Learning Loop	Medium
CAP-TELEMETRY	Telemetry Store	Medium

3.4 Component Dependency Graph

CLI Commands

- └─ depends-on → OpenCode Runner
- └─ depends-on → Telemetry Store
- └─ depends-on → Learning Loop
- └─ depends-on → Prompt Templates

MCP Server

- └─ consumes → Architecture Model API

Learning Loop

- └─ depends-on → Telemetry Store

3.5 Constraints

Constraint	Applies To	Description
CON-TOKENS	MCP Server	Token budget limits on context slicing
CON-NO-HALLUCINATION	CLI Commands	Output must be grounded in repository evidence
CON-TIMEOUT	OpenCode Runner	Execution time limits on runner invocations

Chapter 4

Component Reference Catalog

4.1 Overview

This catalog documents all components in the opencode-arch system, their classifications, responsibilities, and interdependencies.

4.2 Components

4.2.1 COMP-CLI: CLI Commands

Property	Value
ID	COMP-CLI
Kind	module
Layer	LAYER-CLI
Status	ACTIVE

Files: - src/opencode_arch/cli/main.py - src/opencode_arch/cli/extract.py - src/opencode_arch/cli/generate.py - src/opencode_arch/cli/bench.py - src/opencode_arch/cli/docs.py - src/opencode_arch/cli/docs_validator.py - src/opencode_arch/cli/metrics.py - src/opencode_arch/cli/gap_analyzer.py

Responsibilities: - Parse CLI arguments - Orchestrate extraction, generation, regen-loop, docs workflows - Display results to user

4.2.2 COMP-MCP: MCP Server

Property	Value
ID	COMP-MCP

Property	Value
Kind	service
Layer	LAYER-MCP
Status	ACTIVE

Files: - src/opencode_arch/mcp/__main__.py - src/opencode_arch/mcp/server.py - src/opencode_arch/mcp/tools/scan.py - src/opencode_arch/mcp/tools/slice.py - src/opencode_arch/mcp/tools/validate.py - src/opencode_arch/mcp/tools/extract.py - src/opencode_arch/mcp/tools/generate.py

Responsibilities: - Expose architecture tools via MCP protocol - Compress context within token budget - Bridge between LLM agent and architecture-model-standard

4.2.3 COMP-RUNNER: OpenCode Runner

Property	Value
ID	COMP-RUNNER
Kind	library
Layer	LAYER-RUNNER
Status	ACTIVE

Files: - src/opencode_arch/runner/base.py - src/opencode_arch/runner/opencode.py

Responsibilities: - Invoke opencode run as subprocess - Handle timeouts and errors - Return structured RunResult

4.2.4 COMP-LEARNING: Learning Loop

Property	Value
ID	COMP-LEARNING
Kind	module
Layer	LAYER-LEARNING
Status	ACTIVE

Files: - src/opencode_arch/learning/classifier.py - src/opencode_arch/learning/adapter.py - src/opencode_arch/learning/assessor.py - src/opencode_arch/learning/lessons.py - src/opencode_arch/learning/maintainer.py - src/opencode_arch/learning/patterns.py

Responsibilities: - Classify test failure patterns - Adapt prompts based on patterns - Generate report cards with grades - Extract and store lessons - Detect documentation drift

4.2.5 COMP-TELEMETRY: Telemetry Store

Property	Value
ID	COMP-TELEMETRY
Kind	data-store
Layer	LAYER-TELEMETRY
Status	ACTIVE

Files: - src/opencode_arch/telemetry/store.py - src/opencode_arch/telemetry/recorder.py

Responsibilities: - Persist tool invocation metrics - Store regen outcomes and learning data - Provide query interface for metrics display

4.2.6 COMP-PROMPTS: Prompt Templates

Property	Value
ID	COMP-PROMPTS
Kind	library
Layer	LAYER-CLI
Status	ACTIVE

Files: - src/opencode_arch/prompts/regen.py - src/opencode_arch/prompts/extract.py

Responsibilities: - Define LLM prompt templates - Structure system/user prompts for extraction and regen

4.3 Component Dependencies

4.3.1 Direct Dependencies

Source	Target	Relationship
COMP-CLI	COMP-RUNNER	depends-on
COMP-CLI	COMP-TELEMETRY	depends-on
COMP-CLI	COMP-LEARNING	depends-on
COMP-CLI	COMP-PROMPTS	depends-on
COMP-MCP	IF-ARCH-MODEL	consumes
COMP-LEARNING	COMP-TELEMETRY	depends-on

4.3.2 Capability Realizations

Component	Capability	Relationship
COMP-CLI	CAP-EXTRACT	realizes
COMP-CLI	CAP-GENERATE	realizes
COMP-CLI	CAP-DOCS	realizes
COMP-CLI	CAP-REGEN	realizes
COMP-MCP	CAP-MCP	realizes
COMP-LEARNING	CAP-LEARN	realizes
COMP-TELEMETRY	CAP-TELEMETRY	realizes

4.3.3 Constraints Applied

Constraint	Target	Relationship
CON-TOKENS	COMP-MCP	constrained-by
CON-NO-HALLUCINATION	COMP-CLI	constrained-by
CON-TIMEOUT	COMP-RUNNER	constrained-by

4.4 Dependency Summary

COMP-CLI is the primary orchestrator, depending on four other components: COMP-RUNNER for subprocess execution, COMP-TELEMETRY for metrics persistence, COMP-LEARNING for adaptive behavior, and COMP-PROMPTS for LLM prompt construction.

COMP-MCP operates independently of the CLI layer, consuming the external IF-ARCH-MODEL interface directly.

COMP-LEARNING has a single dependency on COMP-TELEMETRY for storing and retrieving pattern data.

COMP-RUNNER, COMP-TELEMETRY, and COMP-PROMPTS are leaf components with no internal dependencies.

Chapter 5

Layer Architecture — opencode-arch

5.1 Overview

The opencode-arch system is organized into a layered architecture with three tiers. Layer ordering determines allowed dependency directions: higher-order layers (infrastructure) are consumed by lower-order layers (user-facing), ensuring clean separation of concerns.

5.2 Layers

5.2.1 Layer 1: CLI Layer (LAYER-CLI)

Property	Value
Order	1 (top — user-facing)
Technology	argparse, asyncio
Directory	src/opencode_arch/cli/
Components	CLI Commands (COMP-CLI), Prompt Templates (COMP-PROMPTS)

The CLI layer is the primary user-facing entry point. It orchestrates extraction, generation, benchmarking, and metrics display workflows by delegating to lower layers.

5.2.2 Layer 1: MCP Server Layer (LAYER-MCP)

Property	Value
Order	1 (top — user-facing)
Technology	FastMCP, mcp-protocol
Directory	src/opencode_arch/mcp/
Components	MCP Server (COMP-MCP)

The MCP Server layer exposes architecture tools to LLM agents via the Model Context Protocol. It operates at the same tier as the CLI — both are entry points, serving different consumers (humans vs. agents).

5.2.3 Layer 2: Learning Layer (LAYER-LEARNING)

Property	Value
Order	2 (middle)
Technology	dataclasses, regex
Directory	src/opencode_arch/learning/
Components	Learning Loop (COMP-LEARNING)

The Learning layer sits between the user-facing entry points and the infrastructure layers. It analyzes telemetry data to derive patterns and improve extraction strategies over time.

5.2.4 Layer 3: Runner Layer (LAYER-RUNNER)

Property	Value
Order	3 (bottom — infrastructure)
Technology	subprocess, Protocol
Directory	src/opencode_arch/runner/
Components	OpenCode Runner (COMP-RUNNER)

The Runner layer provides the execution backend for delegating prompts to the frontier model via subprocess. It implements the RunnerBackend protocol for pluggable model backends.

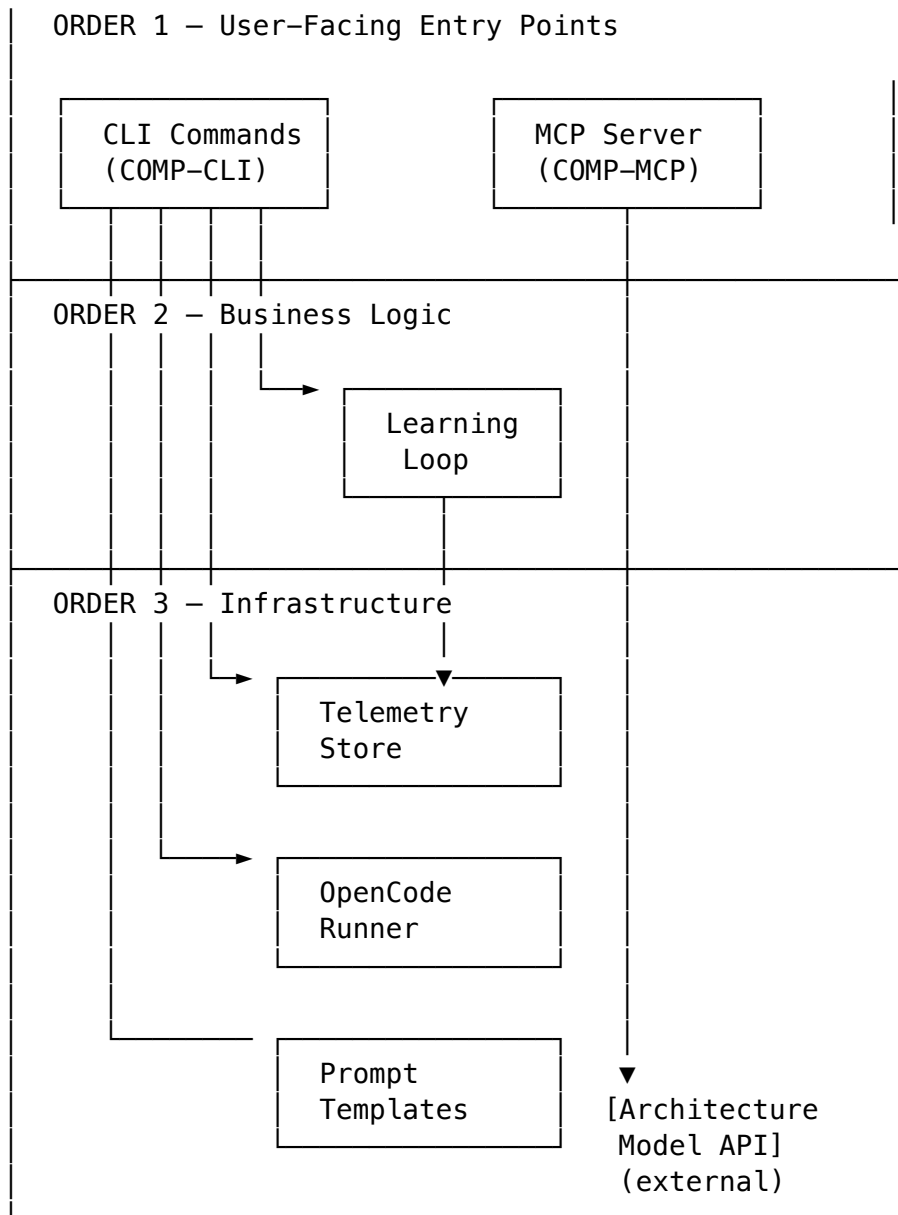
5.2.5 Layer 3: Telemetry Layer (LAYER-TELEMETRY)

Property	Value
Order	3 (bottom — infrastructure)
Technology	SQLite
Directory	src/opencode_arch/telemetry/
Components	Telemetry Store (COMP-TELEMETRY)

The Telemetry layer persists invocation metrics to a local SQLite database. It serves as a shared infrastructure dependency consumed by both the CLI and Learning layers.

5.3 Layer Interaction Map

5.3.1 Dependency Graph



5.3.2 Dependency Summary

Source	Target	Relationship	Cross-Layer
CLI Commands (order 1)	OpenCode Runner (order 3)	depends-on	1 → 3
CLI Commands (order 1)	Telemetry Store (order 3)	depends-on	1 → 3
CLI Commands (order 1)	Learning Loop (order 2)	depends-on	1 → 2
CLI Commands (order 1)	Prompt Templates (order 1)	depends-on	1 → 1 (same layer)

Source	Target	Relationship	Cross-Layer
MCP Server (order 1)	Architecture Model API	consumes	external interface
Learning Loop (order 2)	Telemetry Store (order 3)	depends-on	2 → 3

5.3.3 External Interfaces

The MCP Server consumes the Architecture Model API (IF-ARCH-MODEL), which is an external interface provided by the `architecture-model-standard` package. This is not an internal layer dependency but an external library consumption.

5.4 Capability Realization

Capability	Realized By	Layer
CAP-EXTRACT	CLI Commands	CLI (order 1)
CAP-GENERATE	CLI Commands	CLI (order 1)
CAP-DOCS	CLI Commands	CLI (order 1)
CAP-REGEN	CLI Commands	CLI (order 1)
CAP-MCP	MCP Server	MCP (order 1)
CAP-LEARN	Learning Loop	Learning (order 2)
CAP-TELEMETRY	Telemetry Store	Telemetry (order 3)

5.5 Constraints

Constraint	Applied To	Layer
CON-TOKENS (token budget limits)	MCP Server	MCP (order 1)
CON-NO-HALLUCINATION (grounded output)	CLI Commands	CLI (order 1)
CON-TIMEOUT (execution time limits)	OpenCode Runner	Runner (order 3)

5.6 Layer Ordering Compliance

All dependencies flow in the correct direction (from lower order numbers to higher order numbers, i.e., from user-facing layers toward infrastructure):

- Order 1 → Order 2: CLI depends on Learning Loop — valid downward dependency.
- Order 1 → Order 3: CLI depends on Runner and Telemetry — valid downward dependency (skips order 2, which is acceptable in relaxed layering).
- Order 2 → Order 3: Learning depends on Telemetry — valid downward dependency.
- Order 1 → Order 1: CLI depends on Prompt Templates — valid same-layer dependency.

No layer ordering violations detected. The architecture follows a relaxed layered style where layers may skip intermediate layers when accessing infrastructure. No upward dependencies (higher order → lower order) exist.

Chapter 6

Behavior Flows

6.1 Overview

This document describes the key system behaviors, their triggers, interaction patterns, and step-by-step flows. Each behavior is documented with its trigger conditions, participating actors, preconditions, sequential steps, postconditions, and architectural pattern type.

6.2 BEH-EXTRACT: Extract Architecture

Attribute	Value
Trigger	CLI command <code>opencode-arch extract</code>
Actor	Developer
Pattern	Sequential

6.2.1 Preconditions

- Target repository path is accessible
- Runner backend is configured and available
- `architecture-model-standard` package is installed

6.2.2 Steps

1. Scan repository to generate manifest — Perform AST analysis on the target repository to produce a reality manifest containing modules, functions, classes, imports, and metrics.
2. Format context from manifest within token budget — Compress the manifest into a dense context string that fits within the configured token budget.
3. Send extraction prompt to runner — Assemble the extraction prompt with context and dispatch it to the configured runner backend (e.g., OpenCode subprocess).
4. Parse YAML response into model — Extract the YAML architecture model from the runner's output and parse it into an `ArchitectureModel` structure.

5. Validate model (target score ≥ 80) — Run structural validation checks (ID uniqueness, referential integrity, orphan detection, capability realization, meta completeness) and verify the score meets the target threshold.
6. Store model to `.architecture-model.yaml` — Persist the validated model to the repository root and record telemetry (score, tokens, time).

6.2.3 Postconditions

- `.architecture-model.yaml` exists in the target repository
- Validation score is recorded in telemetry
- Model passes structural validation with score ≥ 80

6.2.4 Sequence Diagram

```
@startuml
title Extract Architecture

actor "Developer" as ACT_DEV
participant "System" as System

ACT_DEV -> System : Scan repository to generate manifest
System -> System : Format context from manifest within token budget
System -> System : Send extraction prompt to runner
System -> System : Parse YAML response into model
System -> System : Validate model (target score  $\geq 80$ )
System -> System : Store model to .architecture-model.yaml

@enduml
```

6.3 BEH-REGEN: Regen-Loop Iteration

Attribute	Value
Trigger	CLI command <code>opcode-arch regen-loop</code>
Actor	Developer
Pattern	Pipeline

6.3.1 Preconditions

- Architecture model (`.architecture-model.yaml`) exists for the target repository
- Test suite is configured and runnable
- Runner backend is configured and available

6.3.2 Steps

1. Load model and decompose into subsystems — Read the architecture model and partition it into independent subsystems for targeted generation.
2. For each subsystem build prompt with model context — Assemble a generation prompt incorporating the relevant model slice as context for the current subsystem.
3. Send prompt to runner (blind or normal mode) — Dispatch the prompt to the runner backend. In blind mode, the runner operates without access to existing source; in normal mode, existing source is included as context.
4. Run subsystem tests — Execute the test suite scoped to the current subsystem to assess code quality.
5. If failing, analyze gaps and build feedback — When tests fail, analyze the failures to identify gaps between generated code and expected behavior, then construct targeted feedback for the next iteration.
6. Iterate until convergence or max iterations — Repeat the generation-test-feedback cycle until all subsystem tests pass or the maximum iteration count is reached.
7. Run full test suite — After all subsystems converge, execute the complete test suite to verify integration correctness.
8. Generate report card — Produce a summary report documenting pass rates, iteration counts, and quality metrics for each subsystem.

6.3.3 Postconditions

- Generated code passes subsystem tests (or max iterations reached)
- Full test suite results are recorded
- Report card summarizing pass rates and iterations is produced
- Telemetry is recorded for each iteration

6.3.4 Sequence Diagram

```
@startuml
title Regen-Loop Iteration

actor "Developer" as ACT_DEV
participant "System" as System

ACT_DEV -> System : Load model and decompose into subsystems
System -> System : For each subsystem build prompt with model context
System -> System : Send prompt to runner (blind or normal mode)
System -> System : Run subsystem tests
System -> System : If failing, analyze gaps and build feedback
System -> System : Iterate until convergence or max iterations
System -> System : Run full test suite
System -> System : Generate report card

@enduml
```

6.4 BEH-DOCS: Generate Documentation

Attribute	Value
Trigger	CLI command <code>opencode-arch docs generate</code>
Actor	Developer
Pattern	Sequential

6.4.1 Preconditions

- Architecture model (`.architecture-model.yaml`) exists for the target repository
- Target repository is accessible for manifest generation
- Runner backend is configured and available

6.4.2 Steps

1. Load architecture model — Read and parse the `.architecture-model.yaml` from the target repository.
2. Generate reality manifest — Perform AST scanning to produce an up-to-date manifest of the repository's actual structure.
3. Select artifacts based on model richness — Determine which documentation artifacts to generate based on the entities and relationships present in the model.
4. For each artifact assemble context — Build a focused context payload for each selected artifact, combining relevant model slices with manifest data.
5. Send generation prompt to runner — Dispatch the documentation generation prompt with assembled context to the runner backend.
6. Write result with frontmatter — Write the generated documentation to disk, prepending structured frontmatter metadata.
7. Generate index.md — Produce an index file linking all generated documentation artifacts.
8. Run validation — Validate the generated documentation for completeness and structural correctness.

6.4.3 Postconditions

- Documentation artifacts are written to the target directory
- Each artifact includes frontmatter metadata
- `index.md` links all generated artifacts
- Validation confirms documentation completeness

6.4.4 Sequence Diagram

```
@startuml
title Generate Documentation

actor "Developer" as ACT_DEV
participant "System" as System
```

```

ACT_DEV -> System : Load architecture model
System -> System : Generate reality manifest
System -> System : Select artifacts based on model richness
System -> System : For each artifact assemble context
System -> System : Send generation prompt to runner
System -> System : Write result with frontmatter
System -> System : Generate index.md
System -> System : Run validation

```

@endum1

6.5 BEH-MCP-SCAN: MCP Scan Tool

Attribute	Value
Trigger	Agent calls <code>architect_scan</code>
Actor	Agent (frontier model)
Pattern	Sequential

6.5.1 Preconditions

- MCP server is running and registered with the agent environment
- Target repository path is valid and accessible

6.5.2 Steps

1. Receive `repo_path` from agent — Accept the repository path parameter from the calling agent via the MCP protocol.
2. Call `generate_manifest` on target — Invoke the AST scanning function from `architecture-model-standard` on the specified repository path.
3. Return manifest dict to agent — Return the resulting manifest dictionary (modules, functions, classes, imports, metrics) to the agent for further reasoning.

6.5.3 Postconditions

- Agent receives a complete manifest dictionary
- No state is persisted (stateless tool invocation)
- Telemetry is recorded for the invocation

6.6 Pattern Summary

Behavior	Pattern	Iteration	Feedback Loop
BEH-EXTRACT	Sequential	Single pass	No
BEH-REGEN	Pipeline	Multi-pass (until convergence)	Yes (test failures → feedback)
BEH-DOCS	Sequential	Single pass per artifact	No
BEH-MCP-SCAN	Sequential	Single pass	No

Chapter 7

Dependency Graph — opencode-arch

7.1 Overview

This document describes the direct dependencies between components in the opencode-arch system, identifies coupling hotspots, and suggests structural improvements.

7.2 Direct Dependencies

7.2.1 Grouped by Source Component

7.2.1.1 COMP-CLI (CLI Commands)

Target	Relationship
COMP-RUNNER (OpenCode Runner)	depends-on
COMP-TELEMETRY (Telemetry Store)	depends-on
COMP-LEARNING (Learning Loop)	depends-on
COMP-PROMPTS (Prompt Templates)	depends-on

7.2.1.2 COMP-LEARNING (Learning Loop)

Target	Relationship
COMP-TELEMETRY (Telemetry Store)	depends-on

7.2.1.3 COMP-MCP (MCP Server)

Target	Relationship
IF-ARCH-MODEL (Architecture Model API)	consumes

7.3 Capability Realization

Component	Capabilities Realized
COMP-CLI	CAP-EXTRACT, CAP-GENERATE, CAP-DOCS, CAP-REGEN
COMP-MCP	CAP-MCP
COMP-LEARNING	CAP-LEARN
COMP-TELEMETRY	CAP-TELEMETRY

7.4 Constraints

Constraint	Applied To
CON-TOKENS (Token budget limits)	COMP-MCP
CON-NO-HALLUCINATION (No hallucinated output)	COMP-CLI
CON-TIMEOUT (Execution timeout)	COMP-RUNNER

7.5 Dependency Analysis

7.5.1 Coupling Metrics

Component	Outgoing Dependencies	Incoming Dependencies	Total Coupling
COMP-CLI	4	0	4
COMP-TELEMETRY	0	2	2
COMP-LEARNING	1	1	2
COMP-RUNNER	0	1	1
COMP-PROMPTS	0	1	1
COMP-MCP	1 (consumes)	0	1

7.5.2 Highly-Coupled Components

COMP-CLI is the highest-coupled component with 4 outgoing dependencies. It acts as the orchestration layer, depending on Runner, Telemetry, Learning, and Prompts to realize its four capabilities (extract, generate, docs, regen).

COMP-TELEMETRY is the most depended-upon component with 2 incoming dependencies (from CLI and Learning). It serves as a shared data sink.

7.5.3 Circular Dependency Analysis

No circular dependencies exist in the current graph. The dependency flow is strictly acyclic:

```

COMP-CLI → COMP-LEARNING → COMP-TELEMETRY
COMP-CLI → COMP-TELEMETRY
COMP-CLI → COMP-RUNNER
COMP-CLI → COMP-PROMPTS
COMP-MCP ..> IF-ARCH-MODEL

```

The graph forms a DAG (directed acyclic graph) with COMP-CLI at the top and COMP-TELEMETRY, COMP-RUNNER, COMP-PROMPTS as leaf nodes.

7.5.4 Observations

1. COMP-CLI is a God Component risk — It depends on 4 other components and realizes 4 capabilities. This high fan-out means changes to any dependency may require CLI changes.
2. COMP-MCP is fully decoupled from COMP-CLI — The MCP server consumes only the external Architecture Model API interface. These two subsystems share no internal dependencies.
3. COMP-TELEMETRY is a stable dependency — It has zero outgoing dependencies, making it a safe foundation for others to depend on.

7.5.5 Suggested Improvements

1. Reduce CLI fan-out — Consider introducing a mediator or orchestration layer between COMP-CLI and its dependencies. Each CLI command could be a thin wrapper that delegates to a use-case object, rather than directly coupling to Runner, Telemetry, Learning, and Prompts.
2. Abstract COMP-TELEMETRY behind an interface — Since two components depend on Telemetry, introducing an interface (IF-TELEMETRY) would allow swapping implementations without affecting dependents.
3. Consider unifying the CLI and MCP paths — Currently COMP-CLI and COMP-MCP are isolated subgraphs. If they share behavioral logic in the future, a shared domain layer would prevent duplication without introducing coupling between them.

Chapter 8

Integration Guide — opencode-arch

8.1 Overview

This document describes the available interfaces, integration patterns, and constraints for developers connecting to the opencode-arch system. It covers both the MCP tool protocol (for LLM agents) and the CLI/library interfaces (for human developers and automation).

8.2 Available Interfaces

Interface	Type	Protocol	Provider	Consumer	Endpoints
IF-CLI	Internal	argparse	CLI Commands	Developer	6
IF-MCP	External	MCP stdio	MCP Server	LLM Agent	5
IF-RUNNER	Internal	subprocess	OpenCode Runner	CLI Commands	1
IF-TELEMETRY	Internal	SQLite	Telemetry Store	CLI Commands	3
IF-ARCH-MODEL	Internal	Python import	OpenCode Runner	CLI Commands	4

8.2.1 IF-MCP — MCP Tool Protocol

- Protocol: MCP stdio (JSON-RPC over stdin/stdout)
- Direction: External — consumed by LLM agents
- Endpoints: 5 tools (architect_scan, architect_slice, architect_validate, architect_extract, architect_generate)
- Data format: JSON request/response per MCP specification

8.2.2 IF-CLI — CLI Interface

- Protocol: argparse

- Direction: Internal — consumed by developers
- Endpoints: 6 commands (extract, generate, bench, metrics, regen-loop, docs)
- Data format: Command-line arguments and flags; output to stdout

8.2.3 IF-RUNNER — Runner Backend Protocol

- Protocol: subprocess
- Direction: Internal — consumed by CLI Commands
- Endpoints: 1 (run)
- Data format: Structured `RunResult` (output string, exit code, success boolean)

8.2.4 IF-TELEMETRY — Telemetry Store Interface

- Protocol: SQLite
- Direction: Internal — consumed by CLI Commands
- Endpoints: 3 (record invocation, query metrics, store regen outcomes)
- Data format: SQLite rows; accessed via Python API

8.2.5 IF-ARCH-MODEL — Architecture Model API

- Protocol: Python import
- Direction: Internal — provided by OpenCode Runner process
- Endpoints: 4 (manifest generation, model parsing, validation, context slicing)
- Data format: Python objects; YAML serialization for persistence

8.3 Integration Patterns

8.3.1 Integrating with the MCP Server (COMP-MCP)

Layer: MCP Server Layer

Technology: FastMCP

Status: ACTIVE

Files: - `src/opencode_arch/mcp/__main__.py` - `src/opencode_arch/mcp/server.py` - `src/opencode_arch/mcp/tools/scan.py` - `src/opencode_arch/mcp/tools/slice.py` - `src/opencode_arch/mcp/tools/validate.py` - `src/opencode_arch/mcp/tools/extract.py` - `src/opencode_arch/mcp/tools/generate.py`

Recommended pattern: Connect via MCP stdio transport. The server exposes five tools that compress repository context within token budgets, bridging between the LLM agent and the architecture-model-standard library. Tools are stateless per-call; the agent orchestrates multi-step workflows (scan → slice → validate → extract).

Responsibilities: - Expose architecture tools via MCP protocol - Compress context within token budget - Bridge between LLM agent and architecture-model-standard

8.3.2 Integrating with CLI Commands (COMP-CLI)

Layer: CLI Layer

Technology: argparse

Status: ACTIVE

Files: - `src/opencode_arch/cli/main.py` - `src/opencode_arch/cli/extract.py` - `src/opencode_arch/cli/generate.py` - `src/opencode_arch/cli/bench.py` - `src/opencode_arch/cli/docs.py` - `src/opencode_arch/cli/docs_validator.py` - `src/opencode_arch/cli/metrics.py` - `src/opencode_arch/cli/gap_analyzer.py`

Recommended pattern: Invoke via the `opencode-arch` command. The CLI orchestrates extraction, generation, regen-loop, and docs workflows by composing the Runner, Telemetry, Learning, and Prompt components. Extend by adding new subcommands in the CLI layer that follow the existing orchestration pattern.

Responsibilities: - Parse CLI arguments - Orchestrate extraction, generation, regen-loop, docs workflows - Display results to user

Dependencies: COMP-RUNNER, COMP-TELEMETRY, COMP-LEARNING, COMP-PROMPTS

8.3.3 Integrating with the OpenCode Runner (COMP-RUNNER)

Layer: Runner Layer

Technology: subprocess

Status: ACTIVE

Files: - `src/opencode_arch/runner/base.py` - `src/opencode_arch/runner/opencode.py`

Recommended pattern: Implement the `RunnerBackend` protocol to provide a custom model backend. The protocol requires a single `run(prompt, repo_path) -> RunResult` method. The default `OpenCodeRunner` invokes `opencode run` as a subprocess. Swap implementations by conforming to the protocol interface defined in `base.py`.

Responsibilities: - Invoke `opencode run` as subprocess - Handle timeouts and errors - Return structured `RunResult`

8.3.4 Integrating with the Learning Loop (COMP-LEARNING)

Layer: Learning Layer

Technology: dataclasses

Status: ACTIVE

Files: - `src/opencode_arch/learning/classifier.py` - `src/opencode_arch/learning/adapter.py` - `src/opencode_arch/learning/assessor.py` - `src/opencode_arch/learning/lessons.py` - `src/opencode_arch/learning/maintainer.py` - `src/opencode_arch/learning/patterns.py`

Recommended pattern: The learning loop is consumed by the CLI layer. It classifies test failure patterns, adapts prompts based on observed patterns, generates report cards with grades, extracts

and stores lessons, and detects documentation drift. Integrate by passing test results through the classifier and using the adapter to refine prompts on subsequent iterations.

Responsibilities: - Classify test failure patterns - Adapt prompts based on patterns - Generate report cards with grades - Extract and store lessons - Detect documentation drift

Dependencies: COMP-TELEMETRY

8.3.5 Integrating with Telemetry (COMP-TELEMETRY)

Layer: Telemetry Layer

Technology: SQLite

Status: ACTIVE

Files: - src/opencode_arch/telemetry/store.py - src/opencode_arch/telemetry/recorder.py

Recommended pattern: Use the TelemetryStore class for direct database access or the record_invocation() async function for fire-and-forget recording. Telemetry persists tool invocation metrics, regen outcomes, and learning data. Query via the store interface for metrics display or analysis.

Responsibilities: - Persist tool invocation metrics - Store regen outcomes and learning data - Provide query interface for metrics display

8.3.6 Integrating with Prompt Templates (COMP-PROMPTS)

Layer: CLI Layer

Technology: Python strings

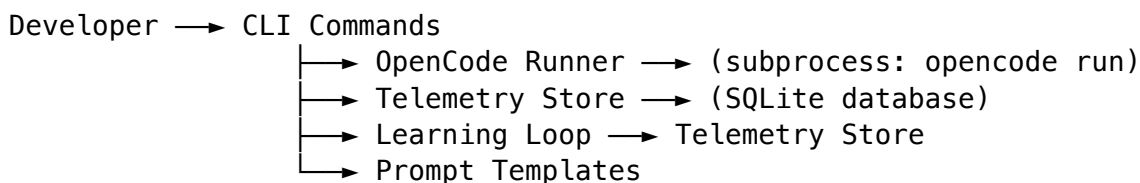
Status: ACTIVE

Files: - src/opencode_arch/prompts/regen.py - src/opencode_arch/prompts/extract.py

Recommended pattern: Import prompt templates directly. Templates define structured system/user prompts for extraction and regen workflows. Customize by extending the template strings or passing additional context variables.

Responsibilities: - Define LLM prompt templates - Structure system/user prompts for extraction and regen

8.4 Component Dependency Graph



LLM Agent → MCP Server → Architecture Model API (Python import)

8.5 Security Constraints & Authentication

8.5.1 CON-PERMISSIONS — Cross-Directory Access

- Type: Security
- Metric: `filesystem_access`
- Requirement: Accessing repositories outside the current working directory requires the `--dangerously-skip-permissions` flag. Without this flag, the system restricts filesystem operations to the immediate project directory.
- Recommendation: Only use this flag in trusted automation environments. Never pass it in user-facing scripts without explicit consent.

8.5.2 CON-NO-HALLUCINATION — No Hallucination Constraint

- Type: Reliability
 - Metric: `grounding_accuracy`
 - Threshold: 100%
 - Requirement: All claims produced by the system must trace back to the model or manifest. The system does not fabricate architecture entities or relationships not grounded in source analysis.
 - Implication for integrators: Do not assume tool outputs contain speculative information. All returned data is derived from AST scanning or validated model content.
-

8.6 Performance Constraints

8.6.1 CON-TOKENS — Token Budget Constraint

- Type: Performance
- Metric: `context_tokens`
- Default threshold: 4000 tokens
- Configurable: Yes — pass budget parameter to `architect_slice`
- Recommendation: Start with the default 4000 token budget. Increase only if extraction quality is insufficient for complex repositories. Monitor via telemetry to find optimal budgets per repository size.

8.6.2 CON-TIMEOUT — Runner Timeout

- Type: Performance
- Metric: `execution_time`
- Threshold: 600 seconds per LLM call
- Requirement: Each invocation of the runner backend (subprocess call to `opcode run`) must complete within 600 seconds. Calls exceeding this threshold are terminated.

- Recommendation: For large repositories, consider reducing the scope of extraction (use focus parameter) rather than increasing timeout.

8.7 Summary of Integration Points

Use Case	Interface	Protocol	Key Constraint
LLM agent consuming architecture tools	IF-MCP	MCP stdio	Token budget (4000 default)
Developer running extractions	IF-CLI	argparse	Runner timeout (600s)
Custom model backend	IF-RUNNER	subprocess	RunnerBackend protocol conformance
Metrics and analytics	IF-TELEMETRY	SQLite	Read-only queries recommended
Extending architecture analysis	IF-ARCH-MODEL	Python import	No hallucination constraint

Chapter 9

API Reference — opencode-arch

9.1 Overview

This document describes the interfaces exposed by the `opencode-arch` package. The system provides five interface boundaries: a CLI for developer interaction, an MCP tool protocol for LLM agents, a pluggable runner backend, a telemetry persistence layer, and an internal dependency on the `architecture-model-standard` library.

9.2 IF-CLI: CLI Interface

Property	Value
ID	IF-CLI
Type	Internal
Protocol	argparse
Provider	COMP-CLI
Consumer	ACT-DEV
Endpoints	6
Data Format	Command-line arguments (positional + optional flags)

9.2.1 Entry Point

```
opencode-arch <command> [options]
```

9.2.2 Endpoints

9.2.2.1 1. **extract**

```
opencode-arch extract <repo_path> [--budget INT] [--focus STR] [--target-score INT] [--model STR] [--timeout INT]
```

Parameter	Type	Required	Default	Description
repo_path	str	yes	—	Path to target repository
--budget	int	no	4000	Token budget for context
--focus	str	no	"all"	Focus scope
--target-score	int	no	80	Minimum validation score
--model	str	no	None	Model override
--timeout	int	no	600	Timeout in seconds

9.2.2.2 2. **generate**

opcode-arch generate <repo_path> [--max-iter INT] [--test-command STR] [--model STR] [--timeout INT]

Parameter	Type	Required	Default	Description
repo_path	str	yes	—	Path to target repository
--max-iter	int	no	3	Maximum iteration attempts
--test-command	str	no	None	Custom test command
--model	str	no	None	Model override
--timeout	int	no	600	Timeout in seconds

9.2.2.3 3. **bench**

opcode-arch bench <repos...> [--output STR] [--model STR]

Parameter	Type	Required	Default	Description
repos	str (nargs=+)	yes	—	One or more repository paths
--output	str	no	None	Output file for metrics JSON
--model	str	no	None	Model override

9.2.2.4 4. **metrics**

opcode-arch metrics [--tool STR] [--last INT] [--learning-curve] [--drift]

Parameter	Type	Required	Default	Description
--tool	str	no	None	Filter by tool name
--last	int	no	10	Number of recent records
--learning-curve	flag	no	False	Show learning curve data
--drift	flag	no	False	Show drift flags

9.2.2.5 5. **report**

opcode-arch report [--repo STR] [--last INT]

Parameter	Type	Required	Default	Description
<code>--repo</code>	str	no	None	Filter by repository
<code>--last</code>	int	no	5	Number of recent reports

9.2.2.6 6. **regen-loop**

`opencode-arch regen-loop --repo STR [--max-iterations INT] [--target FLOAT] [--subsystem STR] [--blind] [--model STR] [--timeout INT]`

Parameter	Type	Required	Default	Description
<code>--repo</code>	str	yes	—	Target repository path
<code>--max-iterations</code>	int	no	5	Maximum regeneration iterations
<code>--target</code>	float	no	0.5	Target pass rate
<code>--subsystem</code>	str	no	None	Specific subsystem to target
<code>--blind</code>	flag	no	False	Run in blind mode
<code>--model</code>	str	no	None	Model override
<code>--timeout</code>	int	no	600	Timeout in seconds

9.3 IF-MCP: MCP Tool Protocol

Property	Value
ID	IF-MCP
Type	External
Protocol	MCP stdio
Provider	COMP-MCP
Consumer	ACT-AGENT
Endpoints	5
Data Format	JSON (MCP tool call/response)

9.3.1 Server Configuration

- Server name: `opencode-arch`
- Transport: `stdio`
- Instructions: “Architecture context compression, validation, and code quality tools”

9.3.2 Endpoints

9.3.2.1 1. **architect_scan**

Generates a reality manifest via AST scanning of the repository.

```
architect_scan(repo_path: str) -> dict
```

Request Parameters:

Parameter	Type	Required	Description
repo_path	str	yes	Absolute path to the repository root

Response Schema:

```
{
  "generated_at": "ISO 8601 timestamp",
  "project_root": "/absolute/path",
  "metrics": { ... },
  "functional_blocks": [ ... ],
  "modules": [ ... ],
  "interfaces": [ ... ]
}
```

Error Response:

```
{
  "error": "Description of what went wrong"
}
```

9.3.2.2 2. **architect_slice**

Compresses repository structure into a dense context string within the token budget.

```
architect_slice(repo_path: str, focus: str = "all", budget: int = 4000, detail: str = "standard")
> str
```

Request Parameters:

Parameter	Type	Required	Default	Description
repo_path	str	yes	—	Absolute path to the repository root
focus	str	no	"all"	Focus scope: "all", F-block ID ("F1"), layer name, or artifact name ("icd")
budget	int	no	4000	Maximum token budget (1 token ≈ 4 chars)
detail	str	no	"standard"	Detail level: "minimal", "standard", or "full"

Response: Plain text string containing compressed context. If `.architecture-model.yaml` exists in the repo, uses the rich `model+ slicer+formatter` path; otherwise falls back to manifest-based compact YAML.

9.3.2.3 3. **architect_validate**

Validates an architecture model YAML for structural correctness.

`architect_validate(model_yaml: str) -> dict`

Request Parameters:

Parameter	Type	Required	Description
<code>model_yaml</code>	<code>str</code>	yes	YAML architecture model string to validate

Response Schema:

```
{
  "score": 85,
  "issues": ["Issue description 1", "Issue description 2"],
  "entity_count": 12,
  "relationship_count": 8,
  "is_valid": true
}
```

Field	Type	Description
<code>score</code>	int (0-100)	Structural correctness score
<code>issues</code>	list[str]	Validation issues found
<code>entity_count</code>	int	Number of entities in model
<code>relationship_count</code>	int	Number of relationships in model
<code>is_valid</code>	bool	Whether model passes validation

9.3.2.4 4. **architect_extract**

Stores a validated architecture extraction to disk and records telemetry.

`architect_extract(repo_path: str, model_yaml: str, context_tokens: int = 0) -> dict`

Request Parameters:

Parameter	Type	Required	Default	Description
<code>repo_path</code>	<code>str</code>	yes	—	Path to the repository root
<code>model_yaml</code>	<code>str</code>	yes	—	YAML architecture model produced by the agent
<code>context_tokens</code>	<code>int</code>	no	0	Tokens of context used (for telemetry)

Response Schema:

```
{
  "stored": true,
  "score": 85,
  "issues": [],
  "path": "/path/to/repo/.architecture-model.yaml",
  "telemetry_recorded": true
}
```

Error Response:

```
{
  "stored": false,
  "error": "Description of what went wrong",
  "score": 0
}
```

9.3.2.5 5. **architect_generate**

Runs the repository's test suite against generated code.

`architect_generate(repo_path: str, test_command: str = "") -> dict`

Request Parameters:

Parameter	Type	Required	Default	Description
<code>repo_path</code>	<code>str</code>	yes	—	Path to the repository with generated code
<code>test_command</code>	<code>str</code>	no	""	Custom test command; defaults to pytest

Response Schema:

```
{
  "passed": true,
  "pass_rate": 0.95,
  "total_tests": 20,
  "passed_tests": 19,
  "failures": ["test_something: AssertionError"]
}
```

Field	Type	Description
<code>passed</code>	<code>bool</code>	Whether all tests passed
<code>pass_rate</code>	<code>float (0.0-1.0)</code>	Ratio of passed to total tests
<code>total_tests</code>	<code>int</code>	Total number of tests discovered
<code>passed_tests</code>	<code>int</code>	Number of tests that passed
<code>failures</code>	<code>list[str]</code>	Descriptions of failed tests

9.4 IF-RUNNER: Runner Backend Protocol

Property	Value
ID	IF-RUNNER
Type	Internal
Protocol	subprocess
Provider	COMP-RUNNER
Consumer	COMP-CLI
Endpoints	1
Data Format	Python Protocol (duck typing)

9.4.1 Protocol Definition

```
class RunnerBackend(Protocol):
    async def run(self, prompt: str, repo_path: str) -> RunResult
```

9.4.2 Endpoint

9.4.2.1 run

Executes an agent with a prompt in the context of a repository.

Parameters:

Parameter	Type	Description
prompt	str	The full prompt to send to the agent
repo_path	str	Absolute path to the target repository

Return Type: **RunResult**

```
@dataclass
class RunResult:
    output: str      # The agent's text output
    exit_code: int   # Process exit code (0 = success)
    success: bool    # Whether the run succeeded
```

9.4.3 Default Implementation: **OpencodeRunner**

Invokes `opencode run` as a subprocess.

Constructor:

Parameter	Type	Default	Description
timeout	int	600	Maximum seconds before timeout
model	str None	None	Model override flag

Subprocess invocation:

opencode run [--model MODEL]

- Prompt passed via stdin (avoids ARG_MAX limits)
- Working directory set to repo_path via cwd
- ANSI escape codes stripped from stdout

9.5 IF-TELEMETRY: Telemetry Store Interface

Property	Value
ID	IF-TELEMETRY
Type	Internal
Protocol	SQLite
Provider	COMP-TELEMETRY
Consumer	COMP-CLI
Endpoints	3
Data Format	SQLite rows / Python dicts

9.5.1 Database Location

Default: ~/.opencode-arch/telemetry.db

9.5.2 Endpoints

9.5.2.1 1. **record**

Records a tool invocation.

```
def record(self, tool: str, repo: str = "", context_tokens: int = 0,
          output_quality: int = 0, iterations: int = 1, metadata: str = "")
```

Parameter	Type	Default	Description
tool	str	—	Tool name (e.g. "extract", "validate")
repo	str	""	Target repository name
context_tokens	int	0	Tokens of context consumed
output_quality	int	0	Validation score (0-100)
iterations	int	1	Number of attempts
metadata	str	""	Additional JSON metadata

Storage schema (**invocations** table):

Column	Type	Description
id	INTEGER	Auto-increment primary key
timestamp	REAL	Unix timestamp

Column	Type	Description
tool	TEXT	Tool identifier
repo	TEXT	Repository name
context_tokens	INTEGER	Tokens used
output_quality	INTEGER	Score 0-100
iterations	INTEGER	Attempt count
metadata	TEXT	JSON string

9.5.2.2 2. **query**

Retrieves recorded invocations with optional filtering.

```
def query(self, tool: str | None = None, limit: int = 100) -> list[dict[str, Any]]
```

Parameter	Type	Default	Description
tool	str None	None	Filter by tool name
limit	int	100	Maximum records to return

Returns: List of dicts matching the invocations table schema, ordered by most recent first.

9.5.2.3 3. **averages**

Computes average metrics for a specific tool.

```
def averages(self, tool: str) -> dict[str, float]
```

Parameter	Type	Description
tool	str	Tool name to compute averages for

Response Schema:

```
{
  "avg_context_tokens": 3200.0,
  "avg_output_quality": 78.5,
  "avg_iterations": 1.3
}
```

9.5.3 Async Wrapper

```
async def record_invocation(store: TelemetryStore, tool: str, repo: str = "",
                           context_tokens: int = 0, output_quality: int = 0,
                           iterations: int = 1, metadata: str = "")
```

Provides an async entry point that delegates to `store.record()`. Designed to be called from async tool handlers without blocking.

9.6 IF-ARCH-MODEL: Architecture Model API

Property	Value
ID	IF-ARCH-MODEL
Type	Internal
Protocol	Python import
Provider	ACT-OPENCODE (architecture-model-standard package)
Consumer	COMP-CLI
Endpoints	4
Data Format	Python objects / YAML strings

9.6.1 Dependency

architecture-model-standard >= 0.3.0

9.6.2 Endpoints

9.6.2.1 1. **generate_manifest()**

AST-scans a repository and produces a structured manifest.

Consumed by: `scan_repository()` in `src/opencode_arch/mcp/tools/scan.py`

Returns: Manifest dict with modules, functions, classes, imports, metrics, and functional blocks.

9.6.2.2 2. **format_model_context()**

Formats an architecture model into a compressed context string for LLM consumption.

Consumed by: `slice_context()` in `src/opencode_arch/mcp/tools/slice.py`

Used when: `.architecture-model.yaml` exists (rich path).

9.6.2.3 3. **validate_model()**

Validates a parsed architecture model for structural correctness. Checks: ID uniqueness, referential integrity, orphan detection, capability realization, meta completeness.

Consumed by: `validate_architecture()` in `src/opencode_arch/mcp/tools/validate.py`

Returns: Score (0-100), list of issues, entity/relationship counts.

9.6.2.4 4. Slicer API

Subsets a model by focus area (F-block, layer, artifact) before formatting.

Consumed by: `_slice_from_model()` in `src/opencode_arch/mcp/tools/slice.py`

Parameters: Model object, focus identifier, budget constraint, detail level.

9.7 Data Contracts

9.7.1 Architecture Model YAML Schema

The canonical data contract between the agent and the storage layer:

```
meta:
  project: string          # Project name
  schema_version: '1.3'   # Schema version

entities:
  components:
    - id: string          # Format: COMP-{N}
      name: string
      status: enum        # ACTIVE | PLANNED | DEPRECATED

  capabilities:
    - id: string          # Format: CAP-{ID}
      name: string
      status: enum        # ACTIVE | PLANNED | DEPRECATED

  layers:
    - id: string          # Format: LAYER-{N}
      name: string
      status: enum

  behaviors:
    - id: string
      name: string

  interfaces:
    - id: string
      name: string

  constraints:
    - id: string
      name: string

  actors:
    - id: string
      name: string

relationships:
  - from: string          # Entity ID reference
    to: string            # Entity ID reference
    type: enum            # realizes | uses | constrains | contains |
                        # triggers | depends_on | implements | exposes
```

9.7.2 MCP Request/Response Contract

All MCP tools follow the standard MCP tool call protocol over stdio:

- Request: JSON-RPC with `tool_name` and `arguments` object
- Response: JSON-RPC with `content` (text or structured data)
- Error: Standard MCP error response with error code and message

9.7.3 RunResult Contract

The data contract between runner backends and the CLI:

Field	Type	Invariants
<code>output</code>	<code>str</code>	Raw text from agent; ANSI codes stripped
<code>exit_code</code>	<code>int</code>	0 = success; -1 = internal error (timeout, not found)
<code>success</code>	<code>bool</code>	True iff <code>exit_code == 0</code>

9.7.4 Telemetry Record Contract

Each tool invocation produces exactly one `invocations` row:

Field	Invariants
<code>timestamp</code>	Set at write time; monotonically increasing
<code>tool</code>	One of: "scan", "slice", "validate", "extract", "generate"
<code>repo</code>	Repository basename or empty string
<code>context_tokens</code>	Non-negative integer
<code>output_quality</code>	Integer 0-100
<code>iterations</code>	Positive integer (minimum 1)

Chapter 10

Constraint Register — opencode-arch

10.1 Overview

This document records the non-functional requirements and design constraints governing the opencode-arch system. Each constraint is identified, classified, measured, and allocated to the component(s) it governs.

10.2 Constraint Definitions

10.2.1 CON-TOKENS — Token Budget Constraint

Attribute	Value
ID	CON-TOKENS
Type	Performance
Metric	context_tokens
Threshold	4000 tokens (default, configurable)
Rationale	The core value proposition of opencode-arch is token arbitrage — compressing full repository structure into minimal context. A hard budget ensures the system delivers on this promise and prevents unbounded token consumption that would negate the efficiency gains of architecture-driven context. The threshold is configurable to allow callers to trade tokens for detail when needed.

10.2.2 CON-NO-HALLUCINATION — No Hallucination Constraint

Attribute	Value
ID	CON-NO-HALLUCINATION
Type	Reliability
Metric	grounding_accuracy
Threshold	100% — all claims must trace to model/manifest
Rationale	Architecture extractions and generated documentation must be grounded exclusively in observable reality (AST manifests, validated models). Any hallucinated entities, relationships, or capabilities would corrupt downstream consumers and erode trust in the system's outputs.

10.2.3 CON-TIMEOUT — Runner Timeout

Attribute	Value
ID	CON-TIMEOUT
Type	Performance
Metric	execution_time
Threshold	600 seconds per LLM call
Rationale	LLM invocations via the runner backend are unbounded by nature. A timeout prevents indefinite blocking during extraction or generation loops, ensures CLI responsiveness, and bounds resource consumption for batch benchmarking scenarios.

10.2.4 CON-PERMISSIONS — Cross-Directory Access

Attribute	Value
ID	CON-PERMISSIONS
Type	Security
Metric	filesystem_access

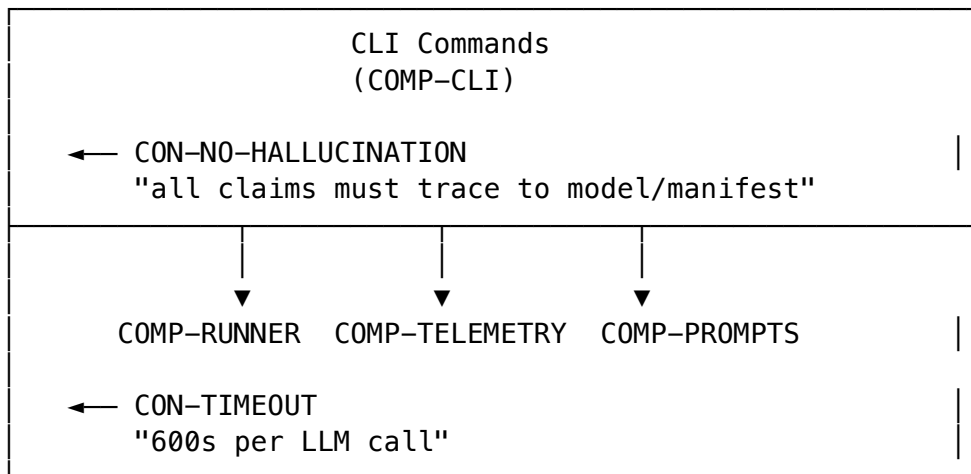
Attribute	Value
Threshold	Requires <code>--dangerously-skip-permissions</code> flag
Rationale	The system scans arbitrary repository paths and writes model files. Cross-directory access is gated behind an explicit opt-in flag to prevent unintended filesystem operations outside the target repository, enforcing the principle of least privilege by default.

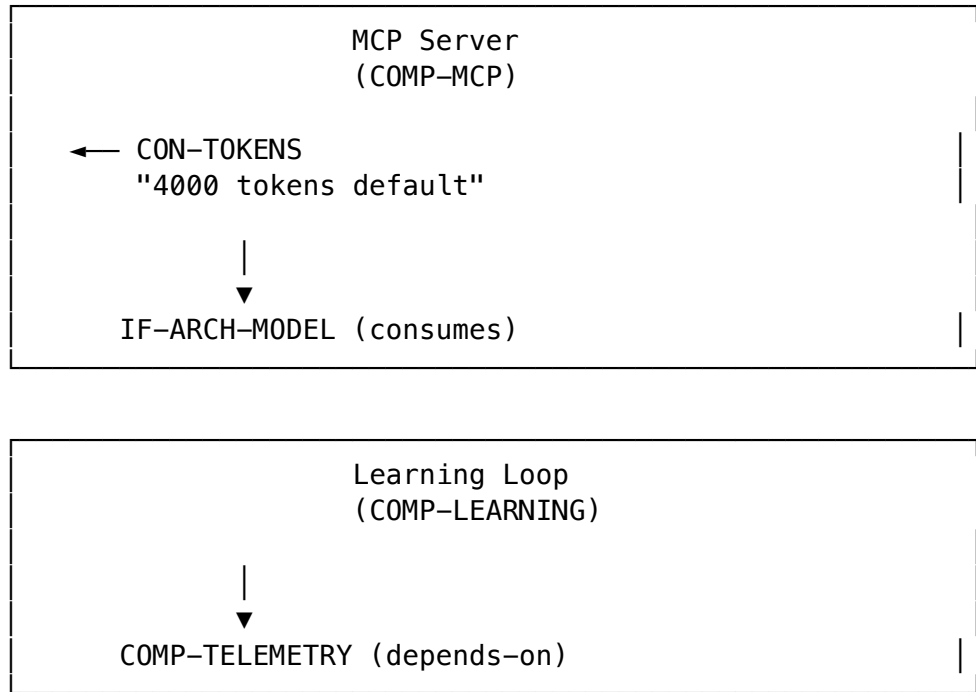
10.3 Constraint Allocation

The following table maps each constraint to the component it governs:

Constraint	Constrained Component	Component Role
CON-TOKENS	COMP-MCP (MCP Server)	Enforces token budget during context slicing
CON-NO-HALLUCINATION	COMP-CLI (CLI Commands)	Ensures extraction outputs trace to manifest/model
CON-TIMEOUT	COMP-RUNNER (OpenCode Runner)	Bounds execution time per LLM subprocess call
CON-PERMISSIONS	(system-wide)	Gates cross-directory filesystem access

10.3.1 Allocation Diagram





10.3.2 Dependency Context

Constraints propagate through the dependency graph. The following dependencies are relevant to constraint enforcement:

- COMP-CLI depends on COMP-RUNNER — timeout constraint (CON-TIMEOUT) on the runner directly affects CLI extraction and generation loops.
- COMP-CLI depends on COMP-TELEMETRY — telemetry records constraint adherence (token usage, execution time).
- COMP-CLI depends on COMP-LEARNING — learning loop consumes telemetry to optimize future token budgets within CON-TOKENS bounds.
- COMP-CLI depends on COMP-PROMPTS — prompt templates are sized to respect CON-TOKENS when combined with context.
- COMP-MCP consumes IF-ARCH-MODEL — token budget (CON-TOKENS) governs how much of the model is materialized into context.
- COMP-LEARNING depends on COMP-TELEMETRY — learning uses recorded metrics to detect constraint violations over time.

10.4 Enforcement Mechanisms

Constraint	Enforcement Point	Mechanism
CON-TOKENS	architect_slice()	budget parameter with default 4000; slicer truncates output to fit

Constraint	Enforcement Point	Mechanism
CON-NO-HALLUCINATION	<code>run_extract()</code> loop	Validation score gates storage; model must parse against manifest
CON-TIMEOUT	<code>OpencodeRunner.run()</code>	Subprocess timeout parameter; raises on expiry
CON-PERMISSIONS	CLI argument parser	Flag must be explicitly passed; default denies cross-directory access

Chapter 11

Deployment View — opencode-arch

11.1 Overview

opencode-arch deploys as two independent entry points — a CLI application and an MCP server — sharing common library layers. All components run in a single Python process (no containers, no network services beyond MCP stdio transport).

11.2 Deployment Units

11.2.1 Unit 1: CLI Application

Layer: CLI

Entry point: opencode-arch command (installed via `pip install -e .`)

Process model: Short-lived subprocess invocations

Component	Kind	Files
COMP-CLI	module	cli/main.py, cli/extract.py, cli/generate.py, cli/bench.py, cli/regen_loop.py, cli/docs.py, cli/docs_validator.py, cli/metrics.py, cli/gap_analyzer.py
COMP-PROMPTS	library	prompts/regen.py, prompts/extract.py

Co-deployed because: Prompt templates are consumed exclusively by CLI orchestration commands. They share the same layer and have no independent deployment lifecycle.

Responsibilities: - Parse CLI arguments and dispatch to subcommands - Orchestrate extraction,

generation, regen-loop, and docs workflows - Structure LLM prompts for extraction and regeneration - Display results to the user

Runtime dependencies: COMP-RUNNER, COMP-TELEMETRY, COMP-LEARNING

11.2.2 Unit 2: MCP Server

Layer: MCP Server

Entry point: `python -m opencode_arch.mcp` (stdio transport)

Process model: Long-running server, one instance per OpenCode session

Component	Kind	Files
COMP-MCP	service	mcp/__main__.py, mcp/server.py, mcp/tools/scan.py, mcp/tools/slice.py, mcp/tools/validate.py, mcp/tools/extract.py, mcp/tools/generate.py

Responsibilities: - Expose 5 architecture tools via MCP protocol (scan, slice, validate, extract, generate) - Compress repository context within token budget - Bridge between LLM agent and architecture-model-standard library

Runtime dependencies: architecture-model-standard package (external), filesystem access to target repositories

11.2.3 Unit 3: OpenCode Runner

Layer: Runner

Deployment: Embedded library (no independent process)

Component	Kind	Files
COMP-RUNNER	library	runner/base.py, runner/opencode.py

Responsibilities: - Invoke `opencode run` as a subprocess - Handle timeouts and errors - Return structured `RunResult` to callers

Consumed by: COMP-CLI (extraction and generation workflows)

11.2.4 Unit 4: Learning Loop

Layer: Learning

Deployment: Embedded module (no independent process)

Component	Kind	Files
COMP-LEARNING	module	learning/classifier.py, learning/adaptor.py, learning/assessor.py, learning/lessons.py, learning/maintainer.py, learning/patterns.py

Responsibilities: - Classify test failure patterns - Adapt prompts based on observed patterns - Generate report cards with grades - Extract and store lessons from outcomes - Detect documentation drift

Runtime dependencies: COMP-TELEMETRY (reads/writes learning data)

11.2.5 Unit 5: Telemetry Store

Layer: Telemetry

Deployment: Embedded data-store (SQLite file)

Storage location: ~/.opencode-arch/telemetry.db

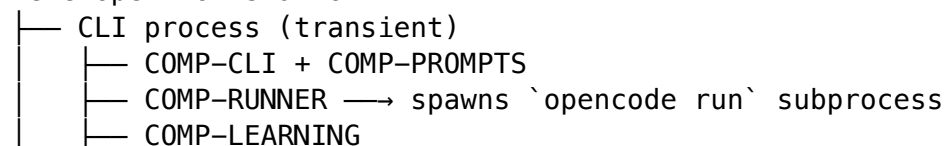
Component	Kind	Files
COMP-TELEMETRY	data-store	telemetry/store.py, telemetry/recorder.py

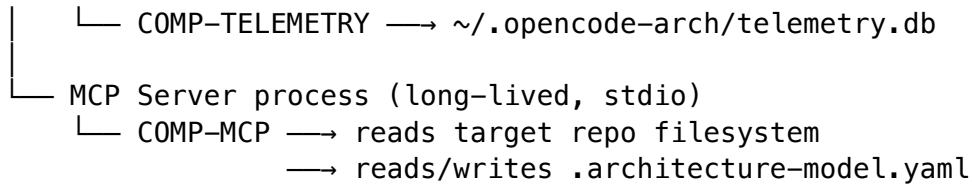
Responsibilities: - Persist tool invocation metrics - Store regen outcomes and learning data - Provide query interface for metrics display

Consumed by: COMP-CLI, COMP-LEARNING

11.3 Deployment Topology

Developer workstation





Both processes run on the same machine. They do not communicate with each other directly — the MCP server is invoked by the LLM agent (via OpenCode), while the CLI is invoked by the developer.

11.4 Operational Constraints

11.4.1 Performance

Constraint	Metric	Threshold	Impact
CON-TOKENS	context_tokens	4000 tokens (default, configurable)	Slice tool must compress full repository structure within budget. Exceeding budget degrades agent reasoning quality and increases cost.
CON-TIMEOUT	execution_time	600 seconds per LLM call	Runner subprocess enforces hard timeout. Long-running extractions are killed and reported as failures.

11.4.2 Reliability

Constraint	Metric	Threshold	Impact
CON-NO-HALLUCINATION	grounding_accuracy	100%	All claims in extracted models must trace back to the manifest or existing model. The slice tool provides only verified context — no fabrication.

11.4.3 Security

Constraint	Metric	Threshold	Impact
CON-PERMISSIONS	filesystem_access	Requires <code>--dangerously-skip-permissions</code> flag	Cross-directory repository access (scanning repos outside the current workspace) requires explicit opt-in via permissions flag.

11.5 Failure Modes

Scenario	Affected Unit	Behavior
SQLite write failure	Telemetry Store	Swallowed — telemetry failures never block tool operation
Runner timeout (>600s)	CLI Application	Subprocess killed, <code>RunResult.success = False</code> returned
Token budget exceeded	MCP Server	Context truncated to fit budget; lower fidelity output
architecture-model-standard not installed	MCP Server	Import error at startup; tools unavailable
Target repo not found	Both	Filesystem error returned to caller

11.6 Prerequisites

- Python 3.11+
- `architecture-model-standard` $\geq$ 0.3.0 installed
- `opencode` CLI available on PATH (for Runner subprocess calls)
- Write access to `~/.opencode-arch/` (telemetry database)
- Read access to target repository filesystem

Chapter 12

Metrics Dashboard

12.1 Code Metrics

Metric	Value
Total Python Files	69
Lines of Code	Not measured
Cyclomatic Complexity	Not measured
Module Count	Not measured

Note: Only file count data is currently available. Lines of code, complexity scores, and module-level breakdowns have not been collected in this measurement pass.

12.2 Quality Assessment

12.2.1 Overall Health

With 69 Python files in the repository, the codebase is moderate in size. No complexity metrics, line counts, or per-module breakdowns are available to assess deeper structural health.

12.2.2 Observations

- File count (69): Suggests a mid-sized project. Without module grouping data, it is unclear how well-organized these files are across packages.
- Missing data: No lines-of-code, complexity, or dependency metrics were collected. A full assessment requires these measurements.

12.2.3 Recommendations

1. Run a static analysis tool (e.g., `radon`, `pylint`, or `ruff`) to collect complexity and line-count metrics.
2. Measure per-module file distribution to identify oversized or undersized packages.
3. Re-generate this dashboard once additional metrics are available.

Dashboard generated from available data. Metrics marked “Not measured” require additional tooling to populate.