RPG & ZeroRepo

Implementation Plan, PRD & Technical Solution Design

Repository Planning Graph for Unified and Scalable Codebase Generation

Based on: Luo, Zhang et al. (Microsoft, October 2025)

With Serena Code Intelligence Integration

Document Version: 1.1 | February 2026

Table of Contents

Part 1: Implementation Plan

1.1 Executive Summary

This document provides an implementation plan and product requirements for building a system based on the Repository Planning Graph (RPG) approach described by Luo, Zhang et al. (Microsoft, October 2025). The paper introduces a structured graph representation that replaces free-form natural language plans for repository-level code generation, demonstrating substantial improvements over existing approaches including Claude Code, Gemini CLI, and Codex.

The core insight is that natural language, whilst flexible, degrades as a planning medium at scale. Its ambiguity causes drift in specifications, fragmented dependencies, and stagnating feature growth. RPG addresses this by encoding repository functionality, file structures, data flows, and function-level interfaces into a persistent, traversable graph.

This revision incorporates Serena (github.com/oraios/serena), an open-source code intelligence toolkit that exposes Language Server Protocol (LSP) capabilities through a Model Context Protocol (MCP) server. Serena addresses a critical blind spot in the RPG paper: all code analysis in the original system is purely LLM-driven, with no traditional static analysis of the generated code. Serena provides ground-truth validation of dependencies, symbol-level localisation, and structural editing — pairing RPG's top-down planning with bottom-up semantic understanding of the actual codebase.

1.2 Paper Analysis: Key Findings

1.2.1 The Problem

Current LLM-based code generation works well at function or file level, but generating entire repositories from scratch remains fundamentally difficult. The gap lies in planning: translating high-level user intent into a coherent network of files, classes, and dependencies. Natural language plans degrade over long horizons, leading to incomplete coverage, overlapping specifications, and brittle designs.

1.2.2 The Proposed Solution

RPG is a graph structure where nodes carry dual semantics. Functionally, they represent progressively refined capabilities (high-level modules down to concrete functions). Structurally, they mirror repository organisation (root nodes align with folders, intermediate nodes with files, leaf nodes with functions/classes). Edges encode inter-module data flows and intra-module ordering, imposing a topological execution order.

ZeroRepo is the framework that constructs and uses the RPG through three stages:

-
-
-

1.2.3 Key Results (RepoCraft Benchmark)

The authors constructed RepoCraft, a benchmark of six real-world Python projects (scikit-learn, pandas, sympy, statsmodels, requests, django) with 1,052 evaluation tasks.

Metric	ZeroRepo (o3-mini)	Claude Code (Strongest Baseline)	Delta
Functionality Coverage	81.5%	54.2%	+27.3pp
Test Pass Rate	69.7%	33.9%	+35.8pp
Voting Rate	75.0%	52.5%	+22.5pp
Lines of Code	~24K	~10.6K	~2.3x
Code Tokens	~261K	~105K	~2.5x
Feature Growth	Near-linear (1,100+ features)	Diminishing returns (~620 features)	~1.8x

With Qwen3-Coder as backbone, ZeroRepo produced ~36K LOC and ~445K tokens, approximately 3.9x larger than Claude Code and 68x larger than other baselines. The graph-guided localisation reduced debugging steps by 30-50% compared to operating without the graph.

1.2.4 Identified Gap: No Static Analysis

A critical limitation of the RPG system as described is that all code analysis is LLM-driven. The system never actually parses or semantically analyses the code it generates. This creates a gap between what the planning graph says the code should contain and what the code actually contains. LLM-based localisation uses fuzzy matching against graph node descriptions rather than real symbol resolution, meaning the system searches its own planning notes rather than the actual codebase. Serena's LSP-powered code intelligence directly addresses this gap.

1.3 Implementation Phases

Phase 1: Foundation (Weeks 1-4)

Establish the core data model, graph structures, basic LLM integration, and Serena code intelligence layer.

- Define the RPG node schema (functionality nodes, folder-augmented nodes, file-augmented nodes, function-augmented nodes).
- Define the RPG edge schema (hierarchical edges, inter-module data flow edges, intra-module ordering edges).
- Implement graph construction primitives: node creation, edge creation, topological sort, subgraph extraction.
- Build the vector database integration for feature tree embedding and retrieval.
- Implement basic LLM interface layer (supporting multiple backends: OpenAI, Anthropic, open-source models).
- Set up Docker-based code execution sandbox for test validation.

-
-

Phase 2: Proposal-Level Construction (Weeks 5-8)

Build the feature planning pipeline that translates user specifications into functionality graphs.

- Integrate or replicate the EpiCoder Feature Tree (or build a comparable feature ontology).
- Implement the explore-exploit subtree selection algorithm (Algorithm 2 from the paper).
- Implement diversity-aware rejection sampling (Algorithm 1) for feature selection.
- Build the LLM-driven candidate filtering and self-check pipeline.
- Implement the refactoring stage that reorganises the subtree into modular subgraphs aligned with the user's repository goal.
- Add iteration tracking and convergence monitoring.

Phase 3: Implementation-Level Construction (Weeks 9-12)

Extend the functionality graph into the full RPG with concrete implementation details.

- Build folder-level encoding: map functional subgraphs to directory namespaces.
- Build file-level encoding: assign features to Python files within folders, with iterative refinement.
- Implement inter-module data flow encoding (DAG construction with typed input/output flows).
- Implement intra-module ordering (file-level dependency ordering within modules).
- Build base class abstraction logic: identify shared patterns and generate abstract interfaces.
- Implement adaptive interface design: cluster leaf features into standalone functions vs. shared classes.
-

Phase 4: Graph-Guided Code Generation (Weeks 13-18)

Build the code generation engine that traverses the RPG to produce working repositories, using Serena for ground-truth localisation and structural editing.

- Implement topological order traversal of the RPG for code generation sequencing.
- Build the test-driven development loop: generate test from docstring, implement function, validate, iterate.
-
-
-

- Implement the staged test validation framework: unit tests, regression tests, integration tests.
- Build majority-vote diagnosis for distinguishing implementation errors from environment/test issues.

Phase 5: Evaluation & Refinement (Weeks 19-22)

Validate the system against benchmarks and iterate on quality.

- Construct evaluation tasks following the RepoCraft methodology (test function harvesting, hierarchical categorisation, stratified sampling).
- Run end-to-end generation on representative repositories.
- Measure coverage, novelty, pass rate, voting rate, and code-level statistics.
- Profile and optimise LLM token usage and iteration counts.
- Refine prompts based on failure analysis.
-

1.4 Key Risks and Mitigations

Risk	Impact	Likelihood	Mitigation
Feature tree unavailability (EpiCoder is not open-source)	High	Medium	Build a custom feature ontology from GitHub topics, Stack Overflow tags, and existing library documentation. Alternatively, use LLM-generated feature trees with manual curation.
LLM cost at scale (30 iterations with multiple LLM calls each)	High	High	Implement caching, batching, and tiered model selection (cheaper models for filtering, stronger models for planning). Budget approximately $5-15 per repository generation.
Prompt sensitivity across different LLM backends	Medium	High	Develop a prompt testing framework with regression tests. Maintain prompt variants per model family.
Topological ordering failures from cyclic dependencies	Medium	Low	Implement cycle detection with automated resolution (dependency inversion, interface extraction). Alert the user when manual intervention is needed.
Test environment fragility	Medium	Medium	Use containerised execution with dependency pinning. Implement the paper's majority-vote diagnosis to distinguish real failures from environment issues.
Scalability beyond Python	Low	Low	Design the RPG schema to be language-agnostic from the start. Language-specific concerns belong in the code generation stage only.

Risk	Impact	Likelihood	Mitigation
Serena language server startup/re-indexing latency	Medium	Medium	Pyright starts quickly, but re-indexing after each file write could add latency. Mitigate with batched writes and Serena's two-tier caching (repeated queries drop from 100-500ms to <10ms).
Serena bootstrap on incrementally generated code	Medium	High	Serena is designed for existing codebases. Implement incremental initialisation: write files to disk before querying, trigger workspace change notifications, handle partial/incomplete code gracefully.

1.5 Resource Estimates

Resource	Estimate	Notes
Engineering team	2-3 senior developers	Experience with LLMs, graph algorithms, software architecture, and LSP/MCP integration required.
Timeline	22 weeks (5.5 months)	Assumes dedicated team. Serena integration adds ~1-2 weeks absorbed into Phases 1 and 4.
LLM API costs (development)	$3,000-$8,000/month	Heavy prompt iteration during phases 2-4. Reduce with caching.
LLM API costs (per generation)	$5-$15 per repository	30 iterations at proposal level + code generation. Varies with repository complexity.
Infrastructure	Docker host, vector database, object storage	Moderate compute requirements. GPU not required for the orchestration layer.
Serena dependency	MIT-licensed, ~17K GitHub stars	Actively maintained (v0.1.4, approaching v1.0). Sponsored by VS Code team and GitHub. Requires Pyright for Python support.

Part 2: Product Requirements Document

2.1 Product Overview

2.1.1 Problem Statement

Software teams and individual developers increasingly want to generate entire repositories from high-level specifications rather than writing code file-by-file. Current approaches (multi-agent frameworks, workflow systems, terminal agents) rely on natural language as the planning medium, which degrades in coherence and stalls in feature growth as repository complexity increases. This results in generated repositories that are small, incomplete, and structurally brittle.

Additionally, existing code generation systems lack ground-truth code intelligence: they generate code but never semantically analyse it, relying entirely on LLM reasoning about code structure. This creates a plan-reality gap where planned dependencies may not match actual dependencies in the generated code.

2.1.2 Product Vision

Build a repository generation system that takes a natural language specification and produces a large-scale, tested, modular Python repository with correct dependencies, data flows, and implementation. The system should combine LLM-driven top-down planning (via RPG) with LSP-driven bottom-up code intelligence (via Serena) to validate and ground generated code in reality.

The system should produce repositories that are 3-5x larger and 2-3x more accurate than current state-of-the-art approaches, with near-linear scaling of features and code size over iterative planning rounds.

2.1.3 Target Users

-
-
-

2.2 Functional Requirements

2.2.1 User Inputs

ID	Requirement	Priority
FR-IN-01	Accept a natural language repository description (free text, 50-5000 words).	Must Have
FR-IN-02	Accept optional constraints: target language (default Python), framework preferences, scope boundaries.	Should Have

ID	Requirement	Priority
FR-IN-03	Accept optional reference materials: API documentation, existing code samples, paper descriptions.	Could Have
FR-IN-04	Support iterative refinement: user can review the RPG at each stage and provide feedback before proceeding.	Should Have

2.2.2 Proposal-Level Planning

ID	Requirement	Priority
FR-PL-01	Generate a functionality graph from the user specification, organised into modular subgraphs.	Must Have
FR-PL-02	Ground feature selection in a structured knowledge base (feature ontology) to ensure stability and diversity.	Must Have
FR-PL-03	Use an explore-exploit strategy to balance precision (known-relevant features) with diversity (novel features).	Must Have
FR-PL-04	Support configurable iteration count for feature selection (default 30, range 5-50).	Should Have
FR-PL-05	Refactor the feature subtree into cohesive, decoupled modules following software engineering principles.	Must Have
FR-PL-06	Report coverage percentage against a reference taxonomy when one is available.	Should Have
FR-PL-07	Identify and flag novel features (those outside the reference taxonomy) separately.	Should Have

2.2.3 Implementation-Level Planning

ID	Requirement	Priority
FR-IL-01	Generate a folder/file skeleton from the functionality graph, mapping modules to directories and files.	Must Have
FR-IL-02	Encode inter-module data flows as typed edges in the RPG (source module, target module, data type, transformation).	Must Have
FR-IL-03	Encode intra-module file ordering to ensure dependency-aware implementation.	Must Have
FR-IL-04	Abstract shared patterns into base classes and common data structures.	Must Have
FR-IL-05	Generate concrete function/class interfaces (signatures, docstrings, type hints) for all leaf nodes.	Must Have
FR-IL-06	Produce a complete RPG that can be serialised, inspected, and modified before code generation.	Must Have

2.2.4 Code Generation

ID	Requirement	Priority
FR-CG-01	Generate code by traversing the RPG in topological order, ensuring dependencies are implemented before dependents.	Must Have
FR-CG-02	Apply test-driven development: generate a test from the task specification, implement the code, validate against the test.	Must Have
FR-CG-03	Provide graph-guided localisation for debugging: RPG-guided search, code view, dependency exploration.	Must Have
FR-CG-04	Support iterative editing: up to 8 debugging iterations per function, up to 20 localisation attempts per iteration.	Should Have
FR-CG-05	Run staged validation: unit tests per function, regression tests on modification, integration tests per completed subgraph.	Must Have
FR-CG-06	Use majority-vote diagnosis (5 rounds) to distinguish implementation errors from environment/test errors.	Should Have
FR-CG-07	Only commit functions that pass all tests to the repository.	Must Have

2.2.5 Code Intelligence (Serena Integration)

ID	Requirement	Priority
FR-CI-01	Provide LSP-powered symbol lookup (find_symbol) as a localisation tool alongside RPG-guided search.	Must Have
FR-CI-02	Validate actual dependencies against RPG-planned dependencies after each code generation step using find_referencing_symbols.	Must Have
FR-CI-03	Provide symbol-level code editing (replace_symbol_body, insert_after_symbol) for surgical modifications during debugging loops.	Should Have
FR-CI-04	Report structural inventory of generated code (get_symbols_overview) for plan-reality comparison at stage boundaries.	Should Have
FR-CI-05	Propagate interface changes (rename_symbol) across the codebase when RPG adaptive interface design modifies shared types.	Should Have
FR-CI-06	Support incremental language server re-indexing as files are generated, without requiring full workspace restart.	Must Have

2.2.6 Outputs

ID	Requirement	Priority
FR-OUT-01	Produce a complete, runnable Python repository with proper directory structure, __init__.py files, setup.py/pyproject.toml.	Must Have
FR-OUT-02	Include generated test suites (unit + integration) alongside source code.	Must Have
FR-OUT-03	Include a README.md describing the repository structure, modules, and usage.	Should Have
FR-OUT-04	Produce the final RPG as an inspectable artefact (JSON/YAML export).	Should Have
FR-OUT-05	Provide a generation report: coverage stats, pass rates, token usage, iteration counts, dependency validation results.	Should Have

2.3 Non-Functional Requirements

ID	Requirement	Target
NFR-01	Repository generation time (end-to-end, medium complexity)	< 60 minutes
NFR-02	Functionality coverage on RepoCraft-equivalent benchmarks	> 75%
NFR-03	Test pass rate on generated repositories	> 60%
NFR-04	Feature growth linearity (R-squared of feature count vs. iteration)	> 0.90
NFR-05	LLM token budget per repository generation	< 5M tokens
NFR-06	System must run on a single machine with 32GB RAM (no GPU required for orchestration)	Required
NFR-07	All generated code must be deterministically reproducible given the same seed and model	Should Have
NFR-08	Serena LSP query latency (cached)	< 10ms
NFR-09	Serena LSP query latency (cold)	< 500ms
NFR-10	Dependency validation accuracy (actual vs. planned match rate)	> 90%

2.4 System Constraints

- The system orchestrates LLM calls but does not train or fine-tune models.
- Code execution for testing must occur in isolated Docker containers to prevent security risks.
- The feature ontology should be extensible by users (custom domain knowledge).

- The RPG schema must be language-agnostic at the graph level, even if initial code generation targets Python.
- All LLM interactions must be logged for debugging and prompt improvement.
- Serena's language server must be configured in read-write mode (not read_only) to enable editing operations during code generation.

Part 3: Technical Solution Design

3.1 Architecture Overview

The system is composed of six principal layers, each with well-defined responsibilities and interfaces. The design follows a pipeline architecture where each stage produces a persistent artefact that the next stage consumes, with the RPG serving as the central data structure that evolves through the pipeline. Serena provides a cross-cutting code intelligence layer that grounds the LLM-driven planning and generation in actual code analysis.

Layer	Responsibility	Key Components
User Interface Layer	Accept specifications, display RPG state, collect feedback, deliver outputs.	CLI tool, optional web dashboard, RPG visualiser.
Orchestration Layer	Coordinate the three-stage pipeline, manage iteration loops, handle errors.	Pipeline controller, iteration manager, state machine.
Planning Layer	Construct and refine the RPG through proposal and implementation stages.	Feature ontology, explore-exploit engine, graph builder, prompt templates.
Generation Layer	Traverse RPG to produce code, run tests, perform localisation and editing.	Code generator, test runner, localisation engine, editing toolkit.
Code Intelligence Layer	Provide LSP-powered code analysis, symbol resolution, dependency validation, and structural editing.	Serena MCP server, Pyright language server, MCP client, symbol cache.
Infrastructure Layer	Provide LLM access, code execution, storage, and vector search.	LLM gateway, Docker sandbox, vector DB, file system.

3.2 Data Model: Repository Planning Graph

3.2.1 Node Schema

Every node in the RPG carries both functional and structural semantics. The node type evolves as the graph progresses through construction stages:

Field	Type	Description
id	UUID	Unique node identifier.
name	String	Human-readable name (e.g., 'Data Loading', 'load_csv').
level	Enum	One of: MODULE, COMPONENT, FEATURE.

Field	Type	Description
node_type	Enum	Evolves through stages: FUNCTIONALITY → FOLDER_AUGMENTED → FILE_AUGMENTED → FUNCTION_AUGMENTED.
parent_id	UUID | null	Reference to parent node (null for root).
folder_path	String | null	Assigned after folder-level encoding (e.g., 'src/data_load').
file_path	String | null	Assigned after file-level encoding (e.g., 'src/data_load/load_data.py').
interface_type	Enum | null	FUNCTION, CLASS, or METHOD. Assigned at leaf level.
signature	String | null	Full function/class signature with type hints.
docstring	String | null	Detailed docstring including args, returns, raises.
implementation	String | null	Actual code, populated during code generation.
test_code	String | null	Generated test code for this node.
test_status	Enum	PENDING, PASSED, FAILED, SKIPPED.
serena_validated	Boolean	Whether Serena has confirmed this node's actual structure matches planned structure.
actual_dependencies	JSON | null	Dependencies discovered by Serena (vs. planned DATA_FLOW edges). Populated during validation.
metadata	JSON	Extensible metadata (feature path from ontology, LLM generation context, etc.).

3.2.2 Edge Schema

Field	Type	Description
id	UUID	Unique edge identifier.
source_id	UUID	Source node.
target_id	UUID	Target node.
edge_type	Enum	HIERARCHY (parent-child), DATA_FLOW (inter-module), ORDERING (intra-module), INHERITANCE, INVOCATION.
data_id	String | null	For DATA_FLOW edges: identifier of the data being passed.
data_type	String | null	For DATA_FLOW edges: type/structure of the data.
transformation	String | null	For DATA_FLOW edges: how data is transformed.

Field	Type	Description
validated	Boolean	Whether Serena has confirmed this edge exists in actual code (for DATA_FLOW and INVOCATION edges).

3.2.3 Graph Operations

The RPG implementation must support the following operations efficiently:

-
-
-
-
-
-
-

3.3 Component Design

3.3.1 Feature Ontology Service

The feature ontology serves as a structured knowledge base for grounding feature selection. The paper uses the EpiCoder Feature Tree (1.5M+ nodes across 7 hierarchical levels). Since this is not publicly available, the implementation should support pluggable ontology backends.

Regardless of approach, each feature node must be embedded into a vector representation (using a sentence embedding model such as text-embedding-3-small or a local model like all-MiniLM-L6-v2) and stored in a vector database (e.g., Qdrant, ChromaDB, or pgvector) with its full hierarchical path as metadata.

3.3.2 Explore-Exploit Engine

This component implements the iterative subtree selection algorithm. Each iteration performs:

1.
2.
3.
4.
5.

The default is 30 iterations with batch sizes tuned to the LLM's context window. The paper reports that coverage typically reaches 70% by iteration 5, 80% by iteration 10, and 95%+ by iteration 30 (Table 3).

3.3.3 Graph Builder

The graph builder converts the functionality subtree into the full RPG through a series of enrichment passes:

3.3.4 Code Generation Engine

The code generation engine traverses the RPG in topological order and implements each leaf node:

1.
2.
3.
4.
5.
6.
7.

3.3.5 Localisation Engine

When a test fails or an edit is needed, the localisation engine provides two complementary tiers of tools:

-
-
-

-
-
-
-

The paper demonstrates (Table 4) that graph-guided localisation alone reduces the mean number of steps from 13.3 to 6.2 for integration testing, 10.8 to 6.8 for incremental development, and 8.5 to 5.8 for debugging. Adding Serena's LSP-powered localisation is expected to reduce these further by eliminating false matches and providing precise type-aware symbol resolution.

The recommended strategy is to attempt Serena localisation first (fast, precise), falling back to RPG-guided search when Serena cannot resolve a symbol (e.g., the code has not yet been written to disk, or the symbol exists only in the planning graph).

3.3.6 Serena Integration Layer

Serena provides LSP-powered code intelligence through 36 MCP tools, built on a custom library called Solid-LSP (derived from Microsoft's multilspy). The integration layer manages the lifecycle of the Serena MCP server and provides a programmatic interface to the RPG system.

Serena Tool	RPG Use Case	Pipeline Stage
find_symbol	Locate exact definition of a class/function during debugging	Stage C: Localisation
find_referencing_symbols	Trace callers and validate actual dependencies	Stage C: Dependency validation
get_symbols_overview	Verify generated structure matches RPG plan	Stages B-C: Verification
replace_symbol_body	Surgically edit a function body without affecting surrounding code	Stage C: Debugging edits
insert_after_symbol / insert_before_symbol	Add new methods or functions at correct positions within classes	Stage C: Code insertion
rename_symbol	Propagate interface name changes across codebase	Stage B: Adaptive interface design
search_for_pattern	Text-level fallback search when LSP resolution fails	Stage C: Localisation fallback
execute_shell_command	Run pytest and capture results within Serena's context	Stage C: Test execution

3.3.7 Test Validation Framework

Testing follows a staged approach aligned with the RPG structure:

-
-
-

All tests execute inside Docker containers. A majority-vote diagnosis (5 rounds, using an LLM judge) classifies failures as either implementation errors (returned for repair) or environment/test errors (handled automatically).

When a test failure occurs, Serena enriches the diagnosis with structural context: find_referencing_symbols maps the failing function's dependencies, identifying what changed recently and surfacing the actual call chain that leads to the error. This transforms the diagnosis from "this function is wrong" to "this function fails because DataLoader.load_csv returns a list but preprocess expects a DataFrame, and three other functions share this dependency."

3.4 LLM Integration Design

3.4.1 Model Selection Strategy

Different stages of the pipeline have different requirements for model capability and cost:

Stage	Recommended Tier	Rationale
Feature filtering (batch)	Mid-tier (e.g., Claude Sonnet, GPT-4o-mini)	High volume, moderate reasoning. Cost-sensitive.
Module refactoring	High-tier (e.g., o3-mini, Claude Sonnet)	Requires strong software architecture reasoning.
File/folder encoding	Mid-tier	Pattern-following task. Well-handled by mid-tier models.
Data flow design	High-tier	Complex multi-module reasoning about dependencies.
Interface design	High-tier	Requires understanding of type systems and API design.
Code generation	High-tier	Core quality-determining step.
Test generation	Mid-tier	Test cases are simpler than implementation.
Failure diagnosis	High-tier	Requires nuanced reasoning about error traces. Serena provides structural context to reduce LLM reasoning burden.

3.4.2 Prompt Architecture

Each LLM interaction uses a structured prompt with the following components:

-
-
-
-
-

The paper provides detailed prompt templates (Appendix A.3, B.1) for each stage. These should be adapted and tested against the chosen LLM backends.

3.5 Technology Stack

Component	Technology	Rationale
Core language	Python 3.11+	Ecosystem compatibility, LLM library support, team familiarity.
Graph library	NetworkX (prototyping) / custom (production)	NetworkX for rapid development; migrate to a custom adjacency-list implementation for performance if needed.
Vector database	ChromaDB (dev) / Qdrant (production)	ChromaDB for local development; Qdrant for scalable, persistent deployment.
Embedding model	text-embedding-3-small or all-MiniLM-L6-v2	Cost-effective for feature similarity. Local model avoids API dependency.
LLM gateway	LiteLLM or custom adapter	Unified interface across OpenAI, Anthropic, and open-source models.
Code intelligence	Serena MCP server (v0.1.4+) with Pyright backend	LSP-powered code analysis, symbol resolution, dependency validation. MIT-licensed, ~17K GitHub stars, actively maintained.
MCP client	Python MCP SDK or custom client	Programmatic invocation of Serena's 36 MCP tools from the orchestration layer.
Code execution	Docker with resource limits	Isolated, reproducible test execution.
Serialisation	JSON (RPG state), SQLite (session persistence)	Simple, inspectable, portable.
CLI framework	Click or Typer	Standard Python CLI tooling.
Testing	pytest	Generated tests use pytest conventions for familiarity.

3.6 Data Flow Diagram

The end-to-end data flow through the system follows this sequence:

1.
2.
3.
4.
5.
6.
7.
8.
9.

3.7 Deployment Architecture

The system is designed to run as a self-contained service or CLI tool. For a production deployment:

-
-
-
-
-

3.8 Serena vs. Alternative Code Intelligence Tools

Several alternative code intelligence MCP servers were evaluated for this integration. Serena was selected for its unique combination of LSP-powered semantic analysis and editing capabilities.

Tool	Approach	Strengths for RPG	Limitations for RPG
Serena (selected)	LSP via Solid-LSP, 36 MCP tools, 30+ languages	Precise symbol resolution, type-aware references, structural editing, two-tier caching, active development.	Designed for existing codebases; requires bootstrap mechanism for incrementally generated code.
GitHub MCP Server	GitHub API-based, repository management	Good for repository operations (commits, PRs).	No semantic code understanding. Text-based search only.
Sourcegraph MCP	Enterprise cross-repo search	Powerful search across large codebases.	Requires Sourcegraph instance. No local semantic operations.
XRAY (ast-grep)	AST pattern matching, 4 languages	Fast structural search.	Limited language support. No type-aware analysis. No editing.

3.9 Open Questions and Decisions Required

Question	Options	Recommendation
Feature ontology source	A) Build custom from public data, B) LLM-generated per request, C) Wait for EpiCoder open-source release	Option A for stability. Supplement with Option B for domain-specific gaps.
Primary LLM backend	A) OpenAI (o3-mini), B) Anthropic (Claude Sonnet/Opus), C) Open-source (Qwen3-Coder)	Start with Option B for development (strong reasoning, good tool use). Test Options A and C for cost optimisation.
Graph persistence	A) In-memory only, B) SQLite, C) Neo4j/graph database	Option B. SQLite provides persistence and queryability without operational complexity. Migrate to C only if graph traversal becomes a bottleneck at very large scale.
Test execution approach	A) Local Docker, B) Remote sandbox service, C) In-process with resource limits	Option A for development and single-user deployment. Option B if offering as a hosted service.
Language support beyond Python	A) Python only (MVP), B) Multi-language from start	Option A. The RPG schema is language-agnostic, but code generation templates are language-specific. Add languages incrementally. Serena's 30+ language support eases future expansion.
Serena invocation method	A) MCP server (subprocess), B) Direct Python API import, C) JetBrains IDE plugin backend	Option A. MCP server provides clean process isolation and standard protocol. Option B would reduce latency but couples tightly. Option C only if JetBrains IDE is available.
Serena re-indexing strategy	A) After every file write, B) Batched (after N files or end of subgraph), C) On-demand (before localisation queries only)	Option B for generation phases (batch writes to reduce overhead). Option C during debugging loops (immediate re-index needed for precise localisation).