CodeKG: A Deterministic Knowledge Graph for Python Codebases with Semantic Indexing and Source-Grounded Snippet Packing

Eric G. Suchanek, PhD
Flux-Frontiers, Liberty Township OH
`suchanek@mac.com`

Abstract

CodeKG constructs a deterministic, auditable knowledge graph from a Python codebase using static analysis. Structural relationships—definitions, calls, imports, and inheritance—are extracted directly from the Python abstract syntax tree and stored in SQLite. A vector index over the same nodes, built with sentence-transformer embeddings and stored in LanceDB, provides semantic retrieval without replacing or overriding the structural record.

The system is organized into four composable layers: a pure AST extractor (`CodeGraph`), a relational graph store (`GraphStore`), a semantic vector index (`SemanticIndex`), and an orchestrator (`CodeKG`) that coordinates the full pipeline and exposes structured result types. A built-in MCP server makes all query and snippet-packing capabilities available as tools to any MCP-compatible AI agent.

The design treats program structure as ground truth and uses semantic embeddings strictly as an acceleration layer. The result is a searchable, traceable representation of a codebase that supports precise navigation, contextual snippet extraction, and downstream reasoning without hallucination.

When coupled with a downstream language model for synthesis, CodeKG constitutes a form of knowledge-graph-augmented generation (KG-RAG). It differs fundamentally from the LLM-constructed graph approach of Microsoft GraphRAG [1]: the knowledge graph is *derived* from formal syntax rather than *inferred* from natural language, making it structurally correct by construction rather than probabilistically approximate. We term this *Structural KG-RAG*.

1 Introduction

As Python systems grow, answering basic architectural questions becomes surprisingly hard. Where is this configuration value actually set? Which functions participate in the connection lifecycle? Where is this service called, and under what conditions? Text search finds strings; IDE symbol lookup finds definitions. Neither tells you much about the shape of the system.

Large language models offer a different kind of help—they can reason about code semantically—but they are not grounded in the source tree. They hallucinate function signatures, invent call relationships, and confidently describe code that does not exist. The problem is not intelligence; it is the absence of a reliable structural substrate.

CodeKG addresses this by building that substrate first. The Python AST is parsed deterministically, producing a graph of nodes (modules, classes, functions, methods) and typed edges (CONTAINS, CALLS, IMPORTS, INHERITS). That graph is stored in SQLite, which is the authoritative record. Semantic embeddings are then built on top of it—not instead of it—so that natural-language queries can find relevant entry points without inventing structure that is not there.

This approach is deliberately inverted relative to systems such as Microsoft GraphRAG [1], which use an LLM to *construct* a knowledge graph from unstructured text. In CodeKG, the LLM has no role in graph construction: the graph is extracted from formal syntax, and the LLM's role is to *reason against* the resulting source-grounded context. Section 13 develops this contrast in detail.

2 Design Principles

Five principles govern the design:

1. **Structure is authoritative.** The AST-derived graph is the source of truth. Embeddings are derived from it and can be discarded and rebuilt at any time without loss.
2. **Semantics accelerate, never decide.** Vector search identifies candidate entry points. Graph traversal determines what is actually returned. The two phases are explicit and separable.
3. **Everything is traceable.** Every node carries a file path and line span. Every edge may carry evidence—a line number and expression text from the call site. Nothing is inferred without a source location.
4. **Determinism over heuristics.** Identical input produces identical output. There are no probabilistic components in the structural layer.
5. **Composable artifacts.** SQLite, LanceDB, Markdown, and JSON are independent outputs. Any downstream consumer—a human, an agent, a CI pipeline—can use whichever format suits it.

3 Architecture

The implementation is organized into four layers. Each has a single responsibility and depends only on layers below it.

3.1 Layer 1 — Primitives (`codekg.py`)

Two frozen dataclasses and one extraction function form the foundation.

`Node` carries: `id` (stable, deterministic), `kind`, `name`, `qualname`, `module_path`, `lineno`, `end_lineno`, and `docstring`.

`Edge` carries: `src`, `rel`, `dst`, and optional `evidence` (a JSON object, typically {`lineno, expr`}).

`extract_repo(repo_root)` walks all `.py` files and runs two deterministic AST passes. Pass 1 extracts definitions and emits `CONTAINS`, `IMPORTS`, and `INHERITS` edges. Pass 2 extracts the call graph, emitting `CALLS` edges with call-site evidence. Unresolved call targets become `sym:` symbol nodes.

Supported node kinds: `module`, `class`, `function`, `method`, `symbol`. Edge relations: `CONTAINS`, `CALLS`, `IMPORTS`, `INHERITS`.

3.2 Layer 2 — `CodeGraph` (`graph.py`)

`CodeGraph` wraps `extract_repo` with lazy caching. The `.nodes` and `.edges` properties trigger extraction on first access; `extract(force=True)` re-runs from scratch. No I/O, no persistence, no embeddings—pure AST extraction with a clean object interface.

3.3 Layer 3 — `GraphStore` (`store.py`)

`GraphStore` manages the SQLite `nodes` and `edges` tables and provides the traversal primitives used by the query layer.

- `write(nodes, edges, wipe=False)` — persist a complete graph via upsert.
- `node(id)` — fetch a single node dict by stable identifier.

- `query_nodes(kinds=, module=)` — filtered node list.
- `edges_within(node_ids)` — edges with both endpoints in the given set.
- `expand(seed_ids, hop=1, rels=...)` — BFS expansion returning `Dict[str, ProvMeta]`.
- `stats()` — node and edge counts by kind and relation.

`ProvMeta` records `best_hop` (minimum hop distance from any seed) and `via_seed` (the originating seed node ID). This provenance is used downstream for ranking.

3.4 Layer 4 — `SemanticIndex` (`index.py`)

`SemanticIndex` reads nodes from a `GraphStore`, constructs a canonical text document for each (name plus docstring), embeds them in batches using a pluggable `Embedder` backend, and stores the vectors in LanceDB. The `search(query, k)` method returns a ranked list of `SeedHit` objects carrying node ID, embedding distance, and rank.

The `Embedder` abstract class defines `embed_texts(texts)` and `embed_query(query)`. The default implementation, `SentenceTransformerEmbedder`, uses `all-MiniLM-L6-v2` from the `sentence-transformers` library (384-dimensional embeddings).

The vector index is derived from SQLite and is fully disposable. Deleting and rebuilding it does not affect the structural record.

3.5 Orchestrator — `CodeKG` (`kg.py`)

`CodeKG` owns all four layers with lazy initialization and exposes the primary user-facing API:

- `build(wipe=False)` — full pipeline: AST extraction, SQLite persistence, LanceDB indexing.
- `build_graph(wipe=False)` — AST extraction and SQLite only.
- `build_index(wipe=False)` — LanceDB indexing only (graph must already exist).
- `query(q, k=8, hop=1, ...)` — hybrid query returning a `QueryResult`.
- `pack(q, k=8, hop=1, ...)` — hybrid query with snippet extraction returning a `SnippetPack`.
- `stats()` — node and edge counts from the store.
- `node(id)` — fetch a single node by ID.

Layer properties (`graph`, `store`, `embedder`, `index`) are lazy: the embedding model and LanceDB connection are only instantiated when first needed. `CodeKG` supports the context manager protocol for clean resource management.

4 Core Data Model

4.1 Nodes

Each node represents a concrete program element extracted from the source tree. The stable identifier is constructed deterministically from kind, module path, and qualified name—for example, `fn:src/code_kg/store.py:GraphStore`
This means node IDs are stable across rebuilds as long as the source structure does not change.

4.2 Edges

Edges encode typed structural relationships. `CALLS` edges carry evidence: a JSON object with the call-site line number and expression text. This makes it possible to extract not just that function A calls function B, but exactly where and in what syntactic context.

5 Build Pipeline

5.1 Static Analysis

`CodeGraph.extract()` invokes `extract_repo()`, which performs two deterministic AST passes over all `.py` files. Pass 1 extracts the definition graph. Pass 2 extracts the call graph with evidence. The resulting nodes and edges are persisted to SQLite via `GraphStore.write()`. This phase uses no embeddings and no language models.

5.2 Semantic Indexing

`SemanticIndex.build(store)` reads `module`, `class`, `function`, and `method` nodes from SQLite, constructs a canonical text document for each, embeds them in configurable batches, and upserts the vectors into LanceDB. Symbol nodes are excluded from the index by default.

Both phases are idempotent. The `--wipe` flag clears existing data before writing; omitting it performs an upsert.

6 Hybrid Query Model

Queries execute in two explicit, separable phases.

Semantic seeding. The query string is embedded and used to retrieve a ranked list of `SeedHit` objects from LanceDB. These nodes serve as conceptual entry points into the graph—the places where the query's meaning most closely matches the codebase's vocabulary.

Structural expansion. `GraphStore.expand()` performs BFS from the seed node IDs, following selected edge types up to a configurable hop limit. Each reachable node is annotated with its minimum hop distance and originating seed via `ProvMeta`. The expansion is bounded and deterministic.

The two phases are independent. Changing the embedding model affects which seeds are selected; changing the hop count or edge filter affects how far the graph is traversed from those seeds. Both can be tuned independently.

7 Ranking and Deduplication

Retrieved nodes are ranked by a composite key: hop distance from seed, seed embedding distance, node kind priority (functions and methods before classes before modules before symbols), and node ID for tie-breaking. The ranking is fully deterministic given fixed inputs.

In `pack()`, nodes are additionally deduplicated by file and source span. A node whose computed span overlaps an already-retained span in the same file is discarded. A configurable cap (`max_nodes`) limits the total result set. This prevents large modules from dominating the output when many nodes from the same file are retrieved.

8 Snippet Packing

For retained nodes, `CodeKG.pack()` extracts source-grounded snippets using the recorded `module_path`, `lineno`, and `end_lineno`. A configurable context window (`context` lines, default 5) is added around each definition span. A per-snippet line cap (`max_lines`, default 160) prevents very large definitions from overwhelming the output. File reads are cached per query. All path resolution goes through a path-traversal-safe join that rejects any path escaping the repository root.

The resulting `SnippetPack` provides `to_markdown()`, `to_json()`, and `save(path, fmt)` methods. The Markdown output includes line numbers, making it suitable for direct ingestion by language models or agents that need to reason about specific source locations.

9 Result Types

Three structured result types are defined in `kg.py`:

- `BuildStats` — returned by all build methods; carries node and edge counts, indexed row count, and embedding dimension.
- `QueryResult` — returned by `query()`; carries the ranked node list, edges within the result set, and query metadata. Provides `to_dict()`, `to_json()`, and `print_summary()`.
- `SnippetPack` — returned by `pack()`; extends the query result with source snippets attached to each node. Provides `to_markdown()`, `to_json()`, and `save()`.

10 MCP Server

`mcp_server.py` wraps `CodeKG` as a Model Context Protocol server, exposing four tools to any MCP-compatible AI agent:

- `graph_stats()` — node and edge counts by kind and relation.
- `query_codebase(q, k, hop, rels, include_symbols)` — hybrid query returning JSON.
- `pack_snippets(q, k, hop, rels, ...)` — hybrid query with source-grounded snippets returning Markdown.
- `get_node(node_id)` — single node metadata lookup by stable ID.

The server is started via the `codekg-mcp` CLI entry point and communicates over `stdio` (for Claude Code, Kilo Code, GitHub Copilot, and Claude Desktop) or `sse` (for HTTP clients). It is strictly read-only; no tool modifies the graph.

11 CLI Entry Points

Six CLI commands are registered as package entry points:

Command	Purpose
`codekg-build-sqlite`	AST extraction → SQLite
`codekg-build-lancedb`	SQLite → LanceDB vector index
`codekg-query`	Hybrid query (JSON output)
`codekg-pack`	Hybrid query + snippet pack (Markdown or JSON)
`codekg-viz`	Launch the Streamlit graph explorer
`codekg-mcp`	Start the MCP server

12 End-to-End Workflow

1. `CodeGraph.extract()` — deterministic AST pass over the repository.
2. `GraphStore.write()` — persist nodes and edges to SQLite.
3. `SemanticIndex.build()` — embed nodes and store vectors in LanceDB.
4. `SemanticIndex.search()` — semantic seeding from a natural-language query.

5. `GraphStore.expand()` — structural BFS expansion with provenance.
6. Deterministic ranking and span-based deduplication.
7. Snippet extraction and `SnippetPack` assembly.

Steps 1–3 are the build pipeline, run once (or on demand after code changes). Steps 4–7 execute on every query, typically in under a second for codebases up to several hundred thousand lines.

13 Comparison with LLM-Constructed Knowledge Graphs

The most directly relevant prior work is GraphRAG, introduced by Edge et al. [1] at Microsoft Research. GraphRAG represents the dominant paradigm in graph-augmented retrieval: an LLM is used to extract entities and relationships from unstructured text documents, those entities are organized into a community hierarchy using graph partitioning, and community-level summaries are pregenerated to support downstream question answering over large corpora.

CodeKG and GraphRAG share a surface resemblance—both augment retrieval with a knowledge graph—but differ in the origin, correctness guarantees, and intended use of that graph.

13.1 Graph Construction: Inferred vs. Derived

In GraphRAG, the knowledge graph is *inferred* by an LLM from natural language text. Entity extraction is probabilistic: the model may miss entities, conflate similar ones, hallucinate relationships, or produce inconsistent descriptions across documents. The graph is only as reliable as the extraction prompt and the LLM's comprehension of each text unit.

In CodeKG, the knowledge graph is *derived* from the Python abstract syntax tree. Every node corresponds to a syntactically verified program element; every edge corresponds to a verified syntactic relationship. A `CALLS` edge from function A to function B exists because a call expression to B was found in the body of A at the recorded line number. The structural layer uses no LLM and contains no probabilistic components. The graph cannot hallucinate.

13.2 The Inversion of the LLM Role

This distinction reverses the role of the language model in the overall pipeline:

- In GraphRAG, the LLM *constructs* the knowledge graph. The graph is a product of LLM inference; its correctness is bounded by LLM accuracy and is not independently verifiable.
- In CodeKG, the LLM *consumes* the knowledge graph. The graph is a product of static analysis; the LLM receives source-grounded snippets with exact file paths and line numbers and reasons against them.

The consequence is a hard boundary on hallucination. In GraphRAG, errors introduced during graph construction propagate silently into query results—the system has no mechanism to detect that an extracted relationship is wrong. In CodeKG, every structural claim is verifiably correct by construction: each node ID encodes its file path and qualified name, and each snippet is extracted from the actual source at the recorded line span.

13.3 Call-Site Evidence

GraphRAG edges carry natural-language descriptions of relationships (e.g., "Entity A collaborates with Entity B in the context of…"), generated by the LLM and approximate. CodeKG `CALLS` edges carry machine-verifiable evidence: the exact line number and expression text of the call site, extracted directly from the AST. This enables a class of query that GraphRAG cannot support: *where exactly is this function called, and in what syntactic context?*

13.4 Different Problem Domains

The two systems address different problems and are not in direct competition.

GraphRAG targets *global sensemaking* over unstructured text corpora. Its design optimises for questions like "What are the main themes in this document collection?" and scales to corpora exceeding one million tokens. It is appropriate when the input is natural language and the goal is synthesis across many documents.

CodeKG targets *structural navigation* of a formal, syntax-governed artifact. It is designed for questions like "Which functions participate in the authentication flow?", "Where is this configuration value read, and which callers depend on it?", and "What is the call graph rooted at this entry point?" It is precise where GraphRAG is approximate, because the input is code—a deterministic, machine-readable artifact—not prose.

13.5 Structural KG-RAG

When CodeKG is coupled with a downstream language model for synthesis, the combined system constitutes knowledge-graph-augmented generation (KG-RAG). The specific flavor, however, is stronger than the general case. Because the knowledge graph is formally derived rather than LLM-inferred, the synthesis stage operates against a *provably correct structural substrate.* Every piece of context passed to the LLM carries a verified file path, exact line span, and—for call relationships—a machine-extracted call-site expression. The LLM's claims are checkable: a node ID such as `fn:src/auth/jwt.py:JWTValidator.validate` at line 47 is a precise, stable, inspectable address.

We term this *Structural KG-RAG*: knowledge-graph-augmented generation in which the graph is extracted from formal syntax rather than inferred from natural language, and in which every piece of retrieved context has deterministic provenance traceable to the source. Table 1 summarises the key distinctions.

Table 1: GraphRAG vs. CodeKG (Structural KG-RAG)

Property	GraphRAG [1]	CodeKG
Graph source	LLM extraction from text	AST (formal syntax)
Graph correctness	Probabilistic	Deterministic
Edge content	NL relationship description	Typed + call-site evidence
Provenance	Document reference	File path + exact line span
LLM role	Builds the graph	Reasons against the graph
Hallucination risk	Present in graph construction	Bounded to synthesis only
Input domain	Unstructured text	Python source code
Query focus	Global sensemaking	Structural navigation
LLM required to build?	Yes	No

14 Conclusion

CodeKG shows that useful, auditable code understanding does not require a language model in the loop for graph construction. Static analysis produces a precise structural record; semantic embeddings make that record navigable with natural language. Keeping the two layers explicit and separable means each can be evaluated, debugged, and improved independently.

The layered architecture—`CodeGraph`, `GraphStore`, `SemanticIndex`, `CodeKG`—cleanly separates concerns and makes the system straightforward to extend. The MCP server turns the query pipeline into a first-class tool for AI agents, giving them grounded, traceable access to codebase structure rather than asking them to reason from memory.

The broader lesson is about where to place trust in an AI-assisted development workflow. Systems that use an LLM to construct the knowledge substrate introduce probabilistic error at the foundation, which propagates unchecked into every downstream reasoning step. CodeKG inverts this: the structural substrate is built from a deterministic formal process, and the LLM is positioned where it is most effective—synthesising and explaining a context pack whose every line is verified against the source. The result is a form of Structural KG-RAG that combines the navigability of natural-language queries with the correctness guarantees of static analysis.

References

[1] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024. `https://arxiv.org/abs/2404.16130`