CodeKG v0:
A Deterministic Knowledge Graph for Python Codebases with Semantic Indexing and Source-Grounded Snippet Packing

Eric G. Suchanek, PhD

Abstract

CodeKG is a system for constructing a deterministic, explainable knowledge graph from a Python codebase using static analysis. The graph captures structural relationships—definitions, calls, imports, and inheritance—directly from the Python abstract syntax tree (AST), stores them in a relational database (SQLite), and augments retrieval with semantic vector indexing (LanceDB).

The system is implemented as four composable, independently testable layers: a pure AST extractor (`CodeGraph`), a relational graph store (`GraphStore`), a semantic vector index (`SemanticIndex`), and a top-level orchestrator (`CodeKG`) that coordinates the full pipeline and exposes structured result types.

Unlike large-language-model–centric approaches, CodeKG treats program structure as ground truth and uses semantic embeddings strictly as an acceleration layer. The result is a searchable, auditable representation of a codebase that supports precise navigation, contextual snippet extraction, and downstream reasoning without hallucination.

1 Introduction

As Python systems scale, answering even basic architectural questions becomes increasingly difficult. Developers often struggle to determine where configuration is defined, how runtime behavior is composed, or where specific services are invoked. Traditional tools such as text search, IDE symbol lookup, and static documentation provide limited support for such higher-level reasoning.

Recent advances in large language models offer semantic intuition but lack grounding in the source tree, leading to brittle or unverifiable conclusions. CodeKG addresses this gap by constructing a first-principles representation of code structure and layering semantic retrieval on top, without sacrificing determinism or provenance.

2 Design Principles

CodeKG v0 is guided by five core principles:

1. **Structure is authoritative.** The AST-derived graph is the source of truth.

2. **Semantics accelerate, never decide.** Vector embeddings are used for ranking and seeding retrieval but do not invent structure.

3. **Everything is traceable.** All nodes and edges map to concrete files and line numbers.

4. **Determinism over heuristics.** Identical input yields identical output.

5. **Composable artifacts.** Relational storage, vector indexing, and human-readable outputs are cleanly separated.

3 Layered Class Architecture

The implementation is organized into four focused layers. Each layer has a single responsibility, is independently testable, and depends only on layers below it.

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

The locked v0 contract. Pure, deterministic, side-effect free. This module is never modified.

- `Node` — frozen dataclass carrying `id`, `kind`, `name`, `qualname`, `module_path`, `lineno`, `end_lineno`, and `docstring`.
- `Edge` — frozen dataclass carrying `src`, `rel`, `dst`, and optional `evidence`.
- `extract_repo(repo_root)` — walks all `.py` files, runs two AST passes (definitions then call graph), and returns `(nodes, edges)`.

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

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

Pure AST extraction with a clean object interface. No I/O, no persistence, no embeddings.

`CodeGraph` wraps `extract_repo` with lazy caching: the `.nodes` and `.edges` properties trigger extraction on first access. Calling `extract(force=True)` re-runs from scratch. The `stats()` method returns node and edge counts by kind.

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

SQLite-backed authoritative store. No embeddings, no AST.

`GraphStore` manages the `nodes` and `edges` tables and provides the graph traversal primitives used by the query layer. Key methods include:

- `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.

The `ProvMeta` object returned by `expand()` records `best_hop` (minimum hop distance from any seed) and `via_seed` (the originating seed node).

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

LanceDB-backed vector index. Derived from SQLite; disposable and rebuildable at any time.

`SemanticIndex` reads nodes from a `GraphStore`, constructs canonical embedding text from names and docstrings, embeds them using a pluggable `Embedder` backend, and stores the vectors in LanceDB. The `search(query, k)` method returns a list of `SeedHit` objects carrying node id, distance, and rank.

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

4 Orchestrator — `CodeKG` (`kg.py`)

`CodeKG` is the top-level entry point. It owns all four layers with lazy initialization and exposes a unified API:

- `build(wipe=False)` — full pipeline: AST extraction, SQLite persistence, LanceDB indexing.
- `build_graph(wipe=False)` — AST extraction and SQLite persistence 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()` — delegate to `GraphStore.stats()`.
- `node(id)` — fetch a single node from the store.

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.

5 Core Data Model

5.1 Nodes

Nodes represent concrete program elements extracted from the source tree. Each node stores a stable deterministic identifier, kind, name, qualified name, module path, source line span, and optional docstring. Nodes are stored in SQLite, which is canonical.

5.2 Edges

Edges encode structural relationships between nodes. Each edge may carry evidence—typically a JSON object containing a source line number and expression text—enabling call-site extraction and precise auditability.

6 Build Pipeline

6.1 Static Analysis Phase

`CodeGraph.extract()` invokes `extract_repo()`, which performs two deterministic AST passes over all `.py` files in the repository. Pass 1 extracts module, class, function, method, and import nodes along with `CONTAINS`, `IMPORTS`, and `INHERITS` edges. Pass 2 extracts the call graph, emitting `CALLS` edges with evidence. Unresolved call targets become `sym:` symbol nodes.

The resulting nodes and edges are persisted to SQLite via `GraphStore.write()`. This phase uses no embeddings and no language models.

6.2 Semantic Indexing Phase

`SemanticIndex.build(store)` reads `module`, `class`, `function`, and `method` nodes from SQLite, constructs a canonical text document for each, embeds them in batches, and upserts the vectors into LanceDB. The vector index is derived and disposable; SQLite remains authoritative.

7 Hybrid Query Model

Queries execute in two explicit phases.

Semantic seeding. The query string is embedded and used to retrieve a ranked list of `SeedHit` objects from the LanceDB index. These nodes serve as conceptual entry points.

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`.

8 Ranking and Deduplication

Retrieved nodes are ranked deterministically by a composite key: hop distance, seed embedding distance, node kind priority (functions and methods before classes before modules before symbols), and node identifier for tie-breaking.

In `pack()`, nodes are additionally deduplicated by file and source span. Nodes whose computed span overlaps an already-retained span in the same file are discarded. A configurable cap limits the total number of returned nodes.

9 Snippet Packing

For retained nodes, `CodeKG.pack()` extracts source-grounded snippets using the recorded `module_path`, `lineno`, and `end_lineno`. Bounded context windows are applied around each definition span. File reads are cached per query. All path resolution is performed through a path-traversal-safe join function.

The resulting `SnippetPack` object provides `to_markdown()`, `to_json()`, and `save(path, fmt)` methods, making it suitable for human review, agent pipelines, or language-model ingestion with grounding.

10 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 `QueryResult` with source snippets attached to each node. Provides `to_markdown()`, `to_json()`, and `save()`.

11 End-to-End Workflow

The overall workflow proceeds as follows:

1. `CodeGraph.extract()` — pure 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 expansion with provenance.

6. Deterministic ranking and span-based deduplication.

7. Snippet extraction and `SnippetPack` assembly.

All steps are coordinated by `CodeKG`, which exposes `build()`, `query()`, and `pack()` as the primary user-facing API.

12 Conclusion

CodeKG demonstrates that explainable, scalable code understanding can be achieved by combining static analysis with semantic indexing while preserving determinism and traceability. The layered class architecture—`CodeGraph`, `GraphStore`, `SemanticIndex`, and `CodeKG`—cleanly separates concerns, enables independent testing of each layer, and makes the system straightforward to extend. By treating structure as ground truth and semantics as an assistive layer, CodeKG provides a robust foundation for navigating and reasoning over complex Python codebases.