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, and augments retrieval with semantic vector indexing.

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 Core Data Model

3.1 Nodes

Nodes represent concrete program elements extracted from the source tree. Supported node kinds include modules, classes, functions, methods, and symbols.

Each node stores a stable identifier, kind, name, qualified name, module path, source line span, and optional docstring. Nodes are stored in a relational database (SQLite) that serves as the canonical representation of the knowledge graph.

3.2 Edges

Edges encode semantic relationships between nodes, including containment, function or method calls, imports, and inheritance. Each edge may carry evidence such as source line numbers or expression text, enabling precise auditability and call-site extraction.

4 Build Pipeline

4.1 Static Analysis Phase

The repository is parsed using Python's abstract syntax tree facilities. All source files are traversed, and definitions, calls, imports, and inheritance relationships are extracted. Normalized node identifiers are generated, and explicit edges are emitted with associated evidence.

The output of this phase is a single relational database containing node and edge tables. This stage uses no embeddings and no language models.

4.2 Semantic Indexing Phase

To support semantic retrieval, a subset of nodes (modules, classes, functions, and methods) is selected for vector indexing. Embedding text is constructed from names and docstrings, embedded using a sentence-transformer model, and stored in a vector database (LanceDB).

The vector index is derived and disposable; the relational graph remains authoritative.

5 Hybrid Query Model

Queries are executed in two phases. First, a natural-language query is embedded and used to retrieve a small set of semantically similar nodes from the vector index. These nodes act as conceptual entry points.

Second, the relational graph is expanded from these seeds using selected edge types. Expansion is bounded by hop count and records provenance such as minimum distance and originating seed.

6 Ranking and Deduplication

Retrieved nodes are ranked deterministically using hop distance, semantic similarity, and node kind priority, favoring executable logic over structural containers.

To prevent redundancy, nodes are deduplicated by file and source span. Overlapping spans within the same file are merged, and per-file limits prevent large modules from dominating results.

7 Snippet Packing

For retained nodes, CodeKG extracts source-grounded snippets using recorded file paths and line spans. Context windows are applied to ensure readability while preserving precision. These snippet packs are suitable for human inspection, agent pipelines, or language-model ingestion with grounding.

8 Call-Site Extraction

Beyond definitions, CodeKG extracts call-site context using evidence stored on call edges. Small source windows around invocation sites are collected, deduplicated, and ranked. This enables precise answers to questions such as where a function is used and under what conditions.

9 End-to-End Workflow

The overall workflow proceeds as follows: repository parsing, relational graph construction, semantic indexing, hybrid query execution, deterministic ranking and deduplication, and finally snippet pack generation.

10 Conclusion

CodeKG demonstrates that explainable, scalable code understanding can be achieved by combining static analysis with semantic indexing while preserving determinism and traceability. By treating structure as ground truth and semantics as an assistive layer, CodeKG provides a robust foundation for navigating and reasoning over complex codebases.