Building an AI-Ready Code Graph RAG based on Clangd Index

Xiao-Feng Li
xli@apache.org
10/10/2025

What does clangd-graph-rag project do?

What: The project ingests clangd index files into a Neo4j graph database.

Code Graph: It builds a code graph with file/folder structure, symbol definitions, and call graph.

Vector index: Has a RAG generation pass enriches the graph with AI-generated summaries and embeddings.

Performance: The pipeline is designed for performance, with parallel data processing and optimized database interactions.

Modular: The system is modular, with different Python scripts responsible for specific passes of the ingestion process.

Compatibility: It can adapt to different clangd indexer versions.

In this document, we give a deep dive into the design and architecture of the clangd-graph-rag project.

Part 1: High-Level Concepts

The Foundation: What is a Clangd Index?

Source of Truth: A Clangd index is a structured dump of the compiler's knowledge about a codebase. It's generated by clangd-indexer, LLVM's language server tool.

Rich Symbol Information: It contains detailed information about every symbol (functions, structs, classes, etc.), including:

- Unique ID (USR)
- Name and Type
- Source Location (declaration and definition)

Reference Data: Crucially, it also indexes every single place a symbol is referenced or used in the code.

Relation Data: For class inheritance etc. Not existing in C-language index

My Goal: To transform this raw, compiler-centric data into a connected knowledge graph that an AI can understand and reason about.

Symbol and Refs

```
--- !Symbol
ID:              BAA4D7A9E4AEF0DA
Name:            free_java_object
Scope:           ''
SymInfo:
  Kind:            Function
  Lang:            C
CanonicalDeclaration:
  FileURI:         'file:///home/xli/NAS/home/bin/mini-jvm/include/gc_for_vm.h'
  Start:
    Line:            17
    Column:          5
  End:
    Line:            17
    Column:          21
Definition:
  FileURI:         'file:///home/xli/NAS/home/bin/mini-jvm/gc/object_create.c'
  Start:
    Line:            61
    Column:          5
  End:
    Line:            61
    Column:          21
Flags:           9
Signature:       '(korp_object *)'
TemplateSpecializationArgs: ''
CompletionSnippetSuffix: '(${1:korp_object *})'
Documentation:   ''
ReturnType:      void
Type:            'c:v'
IncludeHeaders:
  - Header:          'file:///home/xli/NAS/home/bin/mini-jvm/include/gc_for_vm.h'
    References:      2
...
```

```
--- !Refs
ID:              BAA4D7A9E4AEF0DA
References:
  - Kind:            26
    Location:
      FileURI:         'file:///home/xli/NAS/home/bin/mini-jvm/gc/object_create.c'
      Start:
        Line:            61
        Column:          5
      End:
        Line:            61
        Column:          21
    Container:
      ID:              '0000000000000000'
  - Kind:            25
    Location:
      FileURI:         'file:///home/xli/NAS/home/bin/mini-jvm/include/gc_for_vm.h'
      Start:
        Line:            17
        Column:          5
      End:
        Line:            17
        Column:          21
    Container:
      ID:              '0000000000000000'
  - Kind:            28
    Location:
      FileURI:         'file:///home/xli/NAS/home/bin/mini-jvm/natives/java_lang.c'
      Start:
        Line:            132
        Column:          8
      End:
        Line:            132
        Column:          24
    Container:
      ID:              D5AF2A8844BD6186
...
```

Building the Call Graph: The "Easy Way“

The Key Enabler: The Container Field

- Starting from Clangd v21, the index format was improved significantly.
- When a function foo calls another function bar, the reference to bar now includes a Container field pointing to the unique ID of foo.

Direct Graph Construction: This provides a direct, explicit link from a function call (a reference) to its containing function (the caller).

Our Strategy: We simply traverse these links to build the call graph. For every function call reference, we create a [:CALLS] relationship from the Container (caller) to the symbol being referenced (callee).

- This is extremely fast, reliable, and requires no complex analysis.

The Challenge: What If There's No Container Field?

The Problem: Older Clangd versions (and some build systems) do not generate the Container field.

The Gap: The index tells us that function bar was called at file.c:52, but it *doesn't* tell us which function that line of code belongs to. We know the callee, but not the caller.

The Question: How do we spatially map a source code location (file.c:52) to the function that contains it?

The Solution: Tree-sitter

What is Tree-sitter?

- A high-performance, incremental parser generator.
- It builds a concrete syntax tree (CST) that is a detailed, lossless representation of the source code text.
- Unlike an Abstract Syntax Tree (AST), a CST retains all information, including comments, parentheses, and precise source locations.

Our Strategy: Spatial Lookup

- We use tree-sitter to parse every source file in the project.
- From the resulting syntax trees, we extract the precise start and end coordinates (line and column) of every function's body ({...}). This gives us a map of all function boundaries.
- For each function call from the Clangd index, we take its location and perform a spatial search: "Which of the function bodies we just mapped contains this location?"
- Once a containing function is found, we can create the [:CALLS] relationship.

RAG Summaries and the Role of Tree-sitter

The Need for Source Code: To generate a meaningful summary of a function, an LLM needs to see its actual source code.

The Problem: The Clangd index only provides the *location* of a function's definition, not the code itself.

Tree-sitter to the Rescue (Again): We reuse the function body coordinates extracted by tree-sitter.

The RAG Workflow:

- For a given function, get its body coordinates from the tree-sitter data.
- Read those specific lines from the source file on disk.
- Provide the extracted source code to an LLM with a prompt like, "Summarize the purpose of this function."

Conclusion: tree-sitter is a mandatory dependency for this project, as it's essential for RAG generation, even if the call graph can be built without it (using a modern Clangd index).

A Note on RefKind

What is RefKind?: A numeric value in the Clangd index that specifies the *type* of a symbol reference (e.g., declaration, definition, call).

The Change: The numeric values for a function call changed in newer versions of Clangd.

- **Old versions**: A call was Kind: 4, 12 (if spelled).
- **New versions**: A call is Kind: 20, 28 (if spelled).

Our Solution: The call graph builder adaptively checks which kinds to look for based on metadata it infers from the index file itself, making the pipeline resilient to this version change.

RefKind Definition

In clangd-indexer 21.x --------

```
// clang-tools-extra/clangd/index/Ref.h
enum class RefKind : uint8_t {
  Unknown = 0,
  Declaration = 1 << 0, // 1
  Definition = 1 << 1,  // 2
  Reference = 1 << 2,   // 4
  Spelled = 1 << 3,     // 8  means the reference symbol is literally spelled name, not via Macro name
  Call = 1 << 4,        // 16 means this is function reference.
  All = Declaration | Definition | Reference | Spelled,
};
```

In clangd-indexer 16.x --------

```
// clang-tools-extra/clangd/index/Ref.h
enum class RefKind : uint8_t {
  Unknown = 0,
  Declaration = 1 << 0, // 1
  Definition = 1 << 1,  // 2
  Reference = 1 << 2,   // 4
  Spelled = 1 << 3,     // 8  means it is not a MACRO defined name, but literally spelled
  All = Declaration | Definition | Reference | Spelled,
};
```

Part 2: Pipeline Designs

The Full Build Pipeline (clangd_graph_rag_builder.py)

This process builds the entire graph from scratch.

Pass 0: Parse Clangd Index

- The massive YAML index is parsed in parallel into an in-memory collection of Symbol objects. Results are cached to a .pkl file for fast subsequent runs.

Pass 1: Ingest File & Folder Structure

- All unique file/folder paths are discovered from symbol locations.
- :PROJECT, :FOLDER, and :FILE nodes are created in Neo4j, linked by :CONTAINS relationships.

Pass 2: Ingest Symbol Definitions

- :FUNCTION and :DATA_STRUCTURE nodes are created.
- :DEFINES relationships are created from :FILE nodes to the symbols they define.

Pass 3: Ingest Call Graph

- The system adaptively chooses its strategy. If the Container field is present, it uses the direct method. If not, it falls back to the tree-sitter spatial lookup method.

Pass 4 & 5: Cleanup & RAG

- Orphan nodes are removed, and the multi-pass RAG generation process is triggered.

Incremental Update Pipeline (clangd_graph_rag_updater.py)

This process efficiently updates the graph based on a git diff.

Phase 1: Identify Changed Files

- Uses git to get a list of all added, modified, and deleted source files between two commits.

Phase 2: Purge Stale Data

- Deletes nodes and relationships from the graph corresponding to the changed files (e.g., the :FILE node for a deleted file, and the :FUNCTION nodes from a modified file).

Phase 3: Build "Mini-Index"

- This is the key optimization. Instead of re-processing the entire project, we build a small, in-memory index containing only:
  - *Symbols defined in the **added/modified** files.*
  - *The 1-hop neighbors (callers and callees) of those symbols.*

Phase 4: Re-run Ingestion Pipeline

- The standard ingestion passes (1, 2, and 3) are re-run, but only on the tiny "mini-index". This patches the holes in the graph with the new data.

Phase 5: Targeted RAG Update

- The RAG generation process is initiated, starting *only* with the functions that were directly changed. The process then intelligently expands to update contextual summaries of neighbors and roll up changes to parent files and folders.

Part 3: Source Code Architecture

Major Components & Responsibilities

clangd_index_yaml_parser.py (SymbolParser)

- **Role**: The entry point for data.
- **Functionality**: High-speed, parallel YAML parsing. Caches results. Creates the in-memory Symbol object model.

clangd_symbol_nodes_builder.py (PathProcessor, SymbolProcessor)

- **Role**: Builds the graph's structural backbone.
- **Functionality**: Creates :FILE, :FOLDER, :FUNCTION, and :DATA_STRUCTURE nodes. Manages the different strategies for creating :DEFINES relationships.

clangd_call_graph_builder.py (ClangdCallGraphExtractor...)

- **Role**: Builds the behavioral part of the graph.
- **Functionality**: Creates the :CALLS relationships. Contains the logic to adapt between modern and legacy Clangd index formats.

function_span_provider.py (FunctionSpanProvider)

- **Role**: The interface to the source code's physical layout.
- **Functionality**: Uses tree-sitter to get function body coordinates. Decouples the rest of the system from tree-sitter and provides data for both call graph construction and RAG.

code_graph_rag_generator.py (RagGenerator)

- **Role**: The AI enrichment engine.
- **Functionality**: Orchestrates the multi-pass summarization and embedding generation.

Orchestrator Deep Dive

clangd_graph_rag_builder.py (GraphBuilder)

- This class orchestrates the full, top-to-bottom build process.
- Its build() method is a sequence of calls to private methods, each corresponding to a pass in the pipeline (e.g., _pass_0_parse_symbols, _pass_1_ingest_paths).
- It manages the lifecycle of the core data objects, like the SymbolParser and Neo4jManager.

clangd_graph_rag_updater.py (GraphUpdater)

- This class orchestrates the more complex incremental update.
- Its update() method coordinates the multi-phase process: identifying changes, purging the graph, building the "mini-index", re-running a targeted ingestion, and triggering a targeted RAG update.
- It relies on the same core components as the builder, but uses them in a more surgical way.

Part 4: Supporting Modules & Developer Tools

Supporting Modules

neo4j_manager.py

- **Purpose**: A Data Access Layer (DAL) for the Neo4j database.
- **Functionality**: Encapsulates all Cypher queries, manages the database connection, and provides methods for schema creation, data purging, and batch transaction execution.

git_manager.py

- **Purpose**: An abstraction layer over the GitPython library.
- **Functionality**: Provides a clean method (get_categorized_changed_files) to identify added, modified, and deleted files between two commits, which is the foundation of the incremental update process.

llm_client.py

- **Purpose**: A factory for creating clients for various Language Model APIs.
- **Functionality**: Provides a consistent LlmClient interface. Concrete implementations (OpenAiClient, OllamaClient, FakeLlmClient) handle the specifics of each API. This makes the core logic model-agnostic.

input_params.py

- **Purpose**: Centralizes command-line argument definitions.
- **Functionality**: Provides functions that add logical groups of arguments to a parser, ensuring consistency and eliminating duplicate definitions across the multiple executable scripts.

Developer Tools (tools/)

These are simple, standalone scripts created to assist with development, debugging, and direct interaction with the project's dependencies.

get_git_changed_files.py: A CLI wrapper for git_manager to quickly see the categorized file changes between two commits from the command line.

run_cyper_file.py: A utility to execute a .cql file containing one or more Cypher queries against the database. Useful for manual data inspection, debugging, or applying manual patches.

unique_yaml_lines_with_markers.py: A simple parsing tool to help debug and inspect the raw clangd YAML index format.

c_ast_to_dot.py: A utility to visualize the Abstract Syntax Tree (AST) of a C source file. It uses tree-sitter to parse the code and graphviz to render the AST as an image, which is invaluable for debugging parsing logic.

check_if_c_header.py: A helper script that heuristically determines if a .h file is a C or C++ header by checking for sibling C++ files or C++-only keywords in the content. It's used to prevent the C parser from failing on C++ code.

Part 5: Deep Dive into Design & Performance

Design for Reuse: Full vs. Incremental Pipelines

The Challenge: How do you support both a full, from-scratch graph build and a surgical, incremental update without writing the core logic twice?

The Principle: Decouple the **Orchestrators** from the **Processors**.

- **Orchestrators** (GraphBuilder, GraphUpdater) are responsible for *what* data to process.
- **Processors** (SymbolProcessor, RagGenerator, etc.) are responsible for *how* to process the data they are given.

Example 1: SymbolProcessor

- This class ingests symbol data. Its methods operate on a dictionary of Symbol objects.
- In a full build, GraphBuilder passes it the *entire* symbol dictionary from the main parser.
- In an update, GraphUpdater passes it the much smaller dictionary from the "mini-index".
- The SymbolProcessor's code is identical in both cases; it is agnostic to the overall context.

Example 2: RagGenerator

- This class has two entry points: summarize_code_graph() for a full build and summarize_targeted_update() for an incremental one.
- The full build method queries for *all* nodes needing a summary.
- The targeted update method receives a small set of "seed" IDs and intelligently expands the scope just enough to update the affected parts of the graph and its hierarchy.
- Both entry points ultimately use the same underlying worker methods (e.g., _process_one_function...), achieving maximum code reuse.

Performance: Parallelism Strategy

The Principle: Use the right tool for the right job: Processes for CPU-bound tasks and Threads for I/O-bound tasks.

CPU-Bound: YAML Parsing (ProcessPoolExecutor)

- **Task**: Parsing the massive, multi-gigabyte clangd index YAML file is a computationally intensive task.
- **Solution**: The file is split into large, independent chunks in memory. The ProcessPoolExecutor distributes these chunks across multiple CPU cores. Because each process has its own memory space and Python interpreter, this strategy effectively bypasses the Global Interpreter Lock (GIL) and achieves true parallel processing.

I/O-Bound: RAG Generation (ThreadPoolExecutor)

- **Task**: Generating summaries involves making hundreds or thousands of network calls to an LLM API. The program spends most of its time waiting for responses.
- **Solution**: The ThreadPoolExecutor is used to manage a large number of lightweight threads. While one thread is blocked waiting for a network response, the Python GIL is released, allowing other threads to run and send their own network requests. This allows for massive concurrency (e.g., 100+ simultaneous API calls) and maximizes throughput.

Performance: Data Ingestion Strategies

The Challenge: Ingesting millions of :DEFINES relationships can be a major bottleneck. The system provides three strategies, allowing users to choose the best trade-off between speed, safety, and dependencies.

1. unwind-sequential

- **How**: Uses a standard UNWIND clause to process batches of relationships sequentially in a single transaction.
- **Pros**: Simple, 100% idempotent (uses MERGE), and requires no special database plugins.
- **Cons**: Slower than parallel methods for large-scale initial imports.

2. isolated-parallel

- **How**: Groups all relationships by their source :FILE node *before* ingestion. It then uses apoc.periodic.iterate to process these groups in parallel.
- **Pros**: This is the safest parallel strategy. By ensuring all relationships for a given file are in the same unit of work, it guarantees that no two threads will ever try to lock the same :FILE node, completely **eliminating the risk of deadlocks**.

3. batched-parallel (Default)

- **How**: Sends raw batches of relationships directly to apoc.periodic.iterate for parallel processing without any pre-grouping.
- **Pros**: The fastest method, as it avoids the client-side grouping overhead.
- **Cons**: Carries a small, theoretical risk of deadlocks if multiple threads happen to write to the same file node at once. This risk is minimal on a clean build, making it the default for its speed.

Performance: Caching Mechanisms

The Principle: Never do the same expensive work twice. The pipeline uses two distinct caching strategies.

1. Index Parsing Cache (.pkl file)

- **What**: After the initial, slow parse of the clangd YAML file, the resulting in-memory SymbolParser object is serialized to a .pkl file.
- **Validity Check**: On subsequent runs, the script compares the file modification time of the .yaml source file and the .pkl cache file. If the cache is newer, it is loaded directly, skipping the entire parsing step and saving minutes of startup time.

2. Function Span Cache (.function_spans.pkl)

- **What**: The results of running tree-sitter to find all function coordinates are also cached.
- **Validity Check (Git)**: The primary and most robust method. The cache stores the current Git commit hash. The cache is only considered valid if the current commit hash matches the stored one and the working tree is clean (git status).
- **Validity Check (Fallback)**: If the project is not a Git repository, it falls back to comparing the cache file's modification time against the modification times of all source files in the project.

Memory Optimization

The Challenge: The in-memory SymbolParser object, holding the entire project index, can consume many gigabytes of RAM.

The Solution: A careful, phased approach to data handling and garbage collection.

Example: The FunctionSpanProvider as a Cache

- The large SymbolParser object is created and used for the initial graph ingestion passes.
- The FunctionSpanProvider is then initialized. It iterates through the SymbolParser once, extracts *only* the function span data it needs for RAG, and stores it in its own, much smaller internal dictionary.
- Crucially, the FunctionSpanProvider then **sets its internal reference to the SymbolParser to None**.
- This allows the main GraphBuilder orchestrator to safely del the main SymbolParser object.
- Because no other objects hold a reference to it, the Python garbage collector can now free the gigabytes of memory used by the parser *before* the memory-intensive RAG process begins.

Developer Experience Designs

Polymorphic Mocking (FakeLlmClient)

- **Problem**: Calling real LLM APIs is slow and expensive, hindering rapid development and testing.
- **Solution**: A FakeLlmClient was created that conforms to the same base LlmClient interface but simply returns a hardcoded string. The get_llm_client factory returns this client when the user specifies --llm-api fake.
- **Benefit**: This is a clean, polymorphic design. The RagGenerator is completely unaware it is using a fake client; it just calls the generate_summary method. This allows for a powerful debugging/dry-run mode without adding any conditional if/else logic to the core application or production clients.

Centralized Arguments (input_params.py)

- **Problem**: Multiple scripts with shared command-line options led to duplicated code and inconsistencies.
- **Solution**: A dedicated input_params.py module was created to define logical groups of arguments. Each script now declaratively calls functions like add_rag_args(parser) to build its CLI.
- **Benefit**: This ensures consistency, improves maintainability (update an argument in one place), and makes the main scripts cleaner.

Thanks!