Building an AI-Ready Code Graph RAG with Clangd

Deep Dive into the clangd-graph-rag Project

# 1. Project Overview

* What: Ingests clangd index files into a Neo4j graph database.
* Code Graph: Builds graph with file/folder structure, symbol definitions, and call graph.
* Vector Index: RAG generation enriches the graph with AI-generated summaries and embeddings.
* Performance: Designed for speed with parallel processing and optimized database interactions.
* Modular: Modular Python scripts for each ingestion pass.
* Compatibility: Adapts to different clangd indexer versions.

This presentation explores the design and architecture of the clangd-graph-rag project.

# 2. High-Level Concepts

## What is a Clangd Index?

* Source of Truth: Structured dump of compiler knowledge, generated by clangd-indexer.
* Rich Symbol Info: Contains unique ID, name, type, source location, and reference data for all symbols.
* Goal: Transform raw compiler data into a connected knowledge graph for AI reasoning.

## Building the Call Graph (Clangd v21+)

* Key Enabler: Container field links function calls to their callers.
* Strategy: Traverse these links to build the call graph quickly and reliably.
* Implementation: For each call, create a [:CALLS] relationship from caller to callee.

## The Challenge: Missing Container Field

* Older Clangd versions lack the Container field.
* Need to map source code locations to the containing function.
* Question: How to determine the caller function from just a code location?

## Solution: Tree-sitter

* What: High-performance parser generating a concrete syntax tree (CST).
* Strategy: Parse files, extract function boundaries, and map call locations to their containing function.
* Result: Enables creation of [:CALLS] relationships without the Container field.

## RAG Summaries & Tree-sitter

* Need: LLMs require function source code for meaningful summaries.
* Tree-sitter: Provides function body coordinates for code extraction and summarization.
* Conclusion: Tree-sitter is essential for RAG, even if call graph can be built otherwise.

## Note on RefKind

* What: Numeric value indicating the type of symbol reference.
* Change: Values changed between clangd versions (old: 4,12; new: 20,28).
* Solution: Call graph builder adaptively checks kinds based on index metadata.

# 3. Pipeline Designs

## Full Build Pipeline (clangd\_graph\_rag\_builder.py)

1. Parse Clangd Index: Parallel YAML parsing, cache results.
2. Ingest File/Folder Structure: Create :PROJECT, :FOLDER, :FILE nodes and :CONTAINS relationships.
3. Ingest Symbol Definitions: Create :FUNCTION, :DATA\_STRUCTURE nodes and :DEFINES relationships.
4. Ingest Call Graph: Uses direct or tree-sitter method as needed.
5. Cleanup & RAG: Remove orphans, trigger RAG generation.

## Incremental Update Pipeline (clangd\_graph\_rag\_updater.py)

1. Identify Changed Files: Use git to list added/modified/deleted files.
2. Purge Stale Data: Remove nodes/relationships for changed files.
3. Build Mini-Index: Index only changed files and their 1-hop neighbors.
4. Re-run Ingestion: Rerun passes on mini-index to patch the graph.
5. Targeted RAG Update: Start with changed functions and expand contextually.

# 4. Source Code Architecture

## Major Components & Responsibilities

* clangd\_index\_yaml\_parser.py (SymbolParser): Entry point for data; parallel YAML parsing and caching.
* clangd\_symbol\_nodes\_builder.py (PathProcessor, SymbolProcessor): Builds structural backbone; creates nodes and relationships.
* clangd\_call\_graph\_builder.py (ClangdCallGraphExtractor): Builds behavioral graph; adapts to index formats.
* function\_span\_provider.py (FunctionSpanProvider): Interfaces with tree-sitter for function boundaries.
* code\_graph\_rag\_generator.py (RagGenerator): Orchestrates summarization and embedding generation.

## Orchestrator Deep Dive

* clangd\_graph\_rag\_builder.py (GraphBuilder): Orchestrates full build via sequential pipeline passes.
* clangd\_graph\_rag\_updater.py (GraphUpdater): Handles incremental updates with targeted processing.

# 5. Supporting Modules & Developer Tools

## Supporting Modules

* neo4j\_manager.py: Data Access Layer for Neo4j; encapsulates Cypher queries and transactions.
* git\_manager.py: Abstraction over GitPython to identify changed files between commits.
* llm\_client.py: Factory for LLM API clients; model-agnostic interface.
* input\_params.py: Centralizes CLI argument definitions for consistency.

## Developer Tools (tools/)

* get\_git\_changed\_files.py: CLI for git\_manager to display file changes.
* run\_cyper\_file.py: Execute Cypher files for manual database interaction.
* unique\_yaml\_lines\_with\_markers.py: Parses and debugs clangd YAML index.
* c\_ast\_to\_dot.py: Visualizes ASTs using tree-sitter and graphviz.
* check\_if\_c\_header.py: Detects C vs. C++ headers to avoid parsing errors.

# 6. Design & Performance Deep Dive

## Design for Reuse: Full vs. Incremental Pipelines

* Challenge: Support both full and incremental builds without duplicating logic.
* Principle: Orchestrators decide what to process; Processors decide how.
* Examples:
* SymbolProcessor: Operates on symbol dictionaries, agnostic of context.
* RagGenerator: Has full and targeted entry points; underlying worker methods are reused.

## Performance: Parallelism Strategy

* CPU-Bound: YAML parsing uses ProcessPoolExecutor for true parallelism (bypasses GIL).
* I/O-Bound: RAG generation uses ThreadPoolExecutor for concurrent API calls.

## Performance: Data Ingestion Strategies

* unwind-sequential: Simple, idempotent, slower for large imports.
* isolated-parallel (default for updates): Groups by file node to avoid deadlocks, safe parallelism.
* batched-parallel (default for builds): Fastest, minimal deadlock risk during clean builds.

## Performance: Caching Mechanisms

* Index Parsing Cache: Serializes parsed symbols; uses file modification time for validity.
* Function Span Cache: Stores tree-sitter results; validity checked via Git commit hash or file times.

## Memory Optimization

* Phased data handling and garbage collection minimizes RAM usage.
* FunctionSpanProvider: Extracts only needed data, releases SymbolParser for GC before RAG step.

## Developer Experience Designs

* Centralized Arguments: Ensures consistent CLI options and easy updates.
* Polymorphic Mocking (FakeLlmClient): Fast, cheap dev/testing mode with identical interface to production clients.