RAG for Code: Implementation Guide

Building Retrieval-Augmented Generation Systems for Programming

What Makes Code RAG Different?

Code has unique characteristics:

- **Structure matters** - Syntax, indentation, scope
- **Multiple representations** - AST, tokens, dependencies
- **Semantic relationships** - Function calls, imports, inheritance
- **Context dependencies** - Types, variables, modules

Traditional RAG approaches often fall short

Architecture Overview

```
Code Repository
     ↓
Parser & Chunker (AST-aware)
     ↓
Embedding Model (code-specific)
     ↓
Vector Database
     ↓
Retrieval System (hybrid)
     ↓
Context Assembly
     ↓
LLM Generation
```

Challenge 1: Chunking Code

Naive approach:

- Split by character count
- Split by line count
- Loses function boundaries

Better approach:

- Parse into AST
- Chunk by semantic units
- Preserve context

AST-Based Chunking Concept

Parse code into Abstract Syntax Trees (AST), then extract:

- Individual functions with their signatures
- Complete classes with all methods
- Module-level code with imports

Key idea: Let the code's natural structure define boundaries, not arbitrary character/line counts

Chunk Granularity Options

Function-level:

- Individual functions with signatures
- Good for API understanding

Class-level:

- Entire classes with methods
- Preserves OOP relationships

Module-level:

- Full files with imports (Maximum context, but large)

Hybrid:

- Functions + their immediate context

Challenge 2: What to Index?

Beyond just code:

1. **Source code** - The implementation
2. **Docstrings** - Natural language descriptions
3. **Comments** - Developer intent
4. **Type signatures** - Interface contracts
5. **Test cases** - Usage examples
6. **Dependencies** - Import relationships

Multi-Modal Indexing Strategy

For each code chunk, store:

- The actual code implementation
- Natural language documentation
- Function/class signatures with types
- File path and location
- Dependencies (imports, calls)
- Related test cases

This creates a rich, queryable representation

Challenge 3: Embedding Models

Generic embeddings (text-embedding-ada-002):

- Work, but suboptimal for code
- Miss syntactic patterns

Code-specific models:

- **CodeBERT** - Microsoft's code understanding model
- **GraphCodeBERT** - Incorporates data flow
- **UniXcoder** - Multi-language support
- **StarCoder** - Recent, strong performance

Code Embedding Process

1. Tokenize code using code-aware tokenizer
2. Pass through specialized transformer model
3. Extract embedding vector (typically from [CLS] token)
4. Store in vector database

Code-specific models understand syntax, semantics, and common patterns better than general text models

Challenge 4: Hybrid Search

Code retrieval benefits from **multiple signals**:

1. **Semantic similarity** - Vector search
2. **Keyword matching** - BM25, exact matches
3. **Graph relationships** - Call graphs, imports
4. **Metadata filtering** - Language, file type

Hybrid Retrieval Strategy

Combine multiple search approaches:

- **Vector search** finds semantically similar code
- **Keyword search** catches exact matches (function names, variables)
- **Graph search** follows dependencies and relationships

Merge results and re-rank by relevance score

Graph-Based Context Enhancement

Build a code knowledge graph:

- Nodes: Functions, classes, modules
- Edges: Calls, imports, inheritance
- Traverse graph to find related code
- Include neighbors when retrieving context

Example: When retrieving function A, also include functions it calls and functions that call it

Challenge 5: Context Window Management

Code context can be large:

- Function + dependencies
- Class definitions
- Import statements
- Related functions

Strategy: Prioritize what matters most

Context Prioritization Levels

Priority 1: Direct matches (most relevant)

- The functions/classes that match the query

Priority 2: Dependencies

- Functions called by matches
- Import statements

Priority 3: Type definitions

- Custom types and interfaces

Priority 4: Usage examples

- Test cases showing usage

Metadata Enrichment

Add structure for better retrieval:

- Programming language
- Code complexity metrics
- Imported libraries/modules
- Functions called
- Line count, file location
- Last modified date
- Author information

Enables filtering and ranking

Intent-Based Retrieval

Different query types need different retrieval:

"How do I..." → Find examples and documentation

"Fix this error..." → Find error patterns and solutions

"Explain this code..." → Find code + detailed comments

"Implement feature X..." → Find similar implementations

Adapt retrieval strategy to query intent

Test-Driven Context

Tests provide valuable context:

- Show real usage examples
- Demonstrate expected inputs/outputs
- Reveal edge cases and error handling

When retrieving a function, include relevant tests that exercise it

Evaluation Metrics

How to measure code RAG quality:

1. **Retrieval accuracy** - Are correct functions found?
2. **Context relevance** - Is retrieved code useful?
3. **Code correctness** - Does generated code work?
4. **Style consistency** - Matches project patterns?
5. **Response time** - Fast enough for developers?

Advanced: Query Expansion

Enhance queries with technical terms:

- Extract keywords from query
- Add synonyms (e.g., "sort" → "order, arrange, sorted")
- Include related technical terms
- Expand acronyms

Improves retrieval coverage

Advanced: Semantic Code Search

Search by behavior, not just keywords:

- "Find code that validates user input"
- "Show me error handling patterns"
- "Functions that make API calls"

Converts natural language descriptions to embeddings, finds similar code semantically

Lab 7!