Deep Learning - Project Idea

Agentic RAG for Automated Code Documentation

Author: Robin Hefner, Florian Hauptmann, Julian Krauß

29. Oktober 2025

Motivation

Software projects frequently lack high-quality documentation. Writing and maintaining docstrings and module comments is time-consuming, so it is often deferred under deadlines. This leads to outdated or missing documentation, which reduces maintainability and slows onboarding for new team members.

Automating the generation and continuous updating of documentation as code evolves addresses this gap. Large language models (LLMs) can summarize code, but their naive use does not scale. Codebases are large and change frequently, so an LLM cannot ingest the entire repository at once.

Retrieval-Augmented Generation (RAG) mitigates this by fetching relevant code or project knowledge at query time. This conditions the LLM on up-to-date repository context rather than model priors alone. This reduces hallucination risk and avoids retraining.

However, a static RAG pipeline (a fixed 'retrieve-then-generate' step) is limited. It performs a one-shot retrieval and ignores change history and task intent.

Agentic RAG introduces a decision loop where an agent plans multi-step retrieval and generation. The agent decides *what* to retrieve and *when*, and selects which artifacts to regenerate based on recent changes. In this project, agentic RAG means identifying the minimal set of symbols affected by a change and redocumenting only those, plus directly dependent items when necessary. This focuses the LLM on relevant context and reduces compute relative to static RAG. We will also draw on prior work, for example the open-source RepoAgent repository on GitHub, to extract design insights.

Objectives

Design and prototype an agentic RAG system for automated Python code documentation. The system will:

- Parse and index a Python codebase by extracting modules, classes, and functions, then embedding them into a vector index for semantic retrieval.
- Generate documentation with a local LLM (e.g., a strong reasoning model like Qwen2.5 or Code Llama) that produces consistent docstrings and summaries using retrieved code context and a style guide.
- Integrate agentic control that monitors repository changes (e.g., via GitHub Actions) and decides which documentation must be created or updated.

- Support incremental updates so only changed or impacted components are redocumented, not the entire codebase.
- Compare agentic RAG with a static RAG baseline on efficiency and accuracy.

System Architecture

Our system consists of three main components that form a continuous cycle: the **Trigger** (CI Pipeline), the **Knowledge Base** (Vector Database), and the **Core Agent** (the LLM).

1. Code Parser & Indexer (The Knowledge Base)

This component is the agent's 'memory.' Its task is to make the entire repository's state searchable at all times.

- **Code Analysis:** Uses `ast` or `libcst` to parse the Python codebase.
- **Extraction & Chunking:** Identifies and isolates self-contained 'knowledge units': modules, classes (definitions), functions (signatures and code body), and a separate 'Style Guide.'
- **Metadata:** Each unit is enriched with critical metadata (e.g., file path, symbol name, import dependencies).
- **Vectorization:** Each unit is converted into a vector using a pre-trained code embedding model (e.g., `microsoft/CodeBERT-base`).
- **Storage:** All vectors and their metadata are stored and indexed in a vector database (e.g., Qdrant or FAISS).

2. Agentic RAG Controller (The Brain)

This is the core of the system, powered by a single, reasoning-strong LLM (e.g., the `T3Q-qwen2.5-14b` model we identified). This component replaces the rigid 'retrieve-then-generate' pipeline with a dynamic decision loop.

The agent executes a 'Plan-Act-Generate' cycle:

1. **Analysis (Observe):** The agent is notified of a code change (e.g., a `git diff`) by the CI pipeline.
2. **Planning (Think):** The LLM uses its reasoning capabilities to analyze the diff and create a multi-step plan. It decides:
   - Which symbols (functions/classes) were *directly* changed?
   - Which other symbols are *indirectly* affected by these changes (e.g., due to modified signatures)?
   - What documentation must therefore be created or updated?
   - What information do I need *exactly* to complete this task?
3. **Action (Act - RAG):** Based on its plan, the agent *selectively* queries the **Knowledge Base (Component 1)**. It dynamically retrieves context, such as:
   - The exact code of the symbol to be documented.
   - The project's style guide.
   - (Optional) 2-3 examples of already well-documented functions from the database (as 'few-shot' context for style).
4. **Generation (Generate):** With all this collected information (plan + retrieved context), the *same* LLM now formulates the final text—the docstring or module summary.

3. CI-Integration (The Trigger)

This is the starting point for the entire workflow.

- **Technology:** GitHub Actions.
- **Trigger:** Runs on `pull_request` or `merge` to the main branch.
- **Action:** The action identifies the code changes (`git diff`) and passes this information to the **Agentic RAG Controller (Component 2)** to initiate its 'Plan-Act-Generate' cycle.

Methodology

Implementation

- **Code Parsing:** Use Python `ast` or `libcst` to extract modules, classes, functions, signatures, and inline comments.
- **Embedding & Storage:** Create embeddings with a pre-trained code embedding model (e.g., `microsoft/CodeBERT-base`) and store them in FAISS or Qdrant, with metadata such as file path, symbol name, and import/call graph links.
- **Agent Hosting:** Host the **Agentic RAG Controller** (e.g., `T3Q-qwen2.5` via Hugging Face TGI or llama.cpp). The agent's logic will be orchestrated (e.g., with LangChain or lightweight custom rules) to execute the 'Plan-Act-Generate' cycle.
- **CI Integration:** Integrate with GitHub Actions to trigger parsing, indexing, and targeted regeneration on pull requests and merges.

Evaluation

We will evaluate the system by comparing our Agentic RAG approach against a static 'regenerate everything' baseline. The key criteria are:

- **Efficiency:** We will measure and compare:
  - Total LLM calls and tokens processed (as a measure of cost).
  - End-to-end processing time (as a measure of speed).
  - We hypothesize the agentic approach will be significantly more efficient for small, common code changes.
- **Adaptability (The 'Agentic' Test):** We will test the system's decision-making with specific change types:
  - *Internal Logic Change:* Test if the agent correctly does *nothing* when a function's signature is unchanged (avoiding 'wasted operations').
  - *Interface (Signature) Change:* Test if the agent correctly identifies and updates not only the changed function but also all *dependent* functions that call it (avoiding 'false negatives').
- **Quality (If time allows):** We will use a manual rubric to assess:
  - Accuracy (e.g., correct parameters, returns, and raised errors).
  - Adherence to the project's style guide.