RAG and MCP integration

Tbd

Technical guide

Setting up a RAG (Retrieval-Augmented Generation) system with MCP (Model Context Protocol) is a powerful way to give your AI coding assistants deep, contextual understanding of your entire project. The core idea is to create a searchable "memory" for your codebase that Claude and Qwen can query on demand.

The table below breaks down the key components and technology choices for this setup.

Component	Role & Purpose	Recommended Technology / Tools
Vector Database	Stores numerical representations (embeddings) of your code, enabling fast semantic search.	**Milvus** (via Zilliz Cloud) or **ChromaDB**
Embedding Model	Converts your code snippets and text into numerical vectors that capture their meaning.	**Qwen3-Embedding** models or general-purpose models like `all-MiniLM-L6-v2`
MCP Server (The Bridge)	A custom server that uses the vector database to perform RAG. It exposes this capability as a "tool" that AI CLIs can call.	A custom Python script using frameworks like **FastMCP** and **LangChain**
AI CLI Client	Your interface (Claude Code or Qwen Code) configured to connect to the MCP server.	**Claude Code**, **Qwen Code**, or IDEs like **Cursor**

Here is a step-by-step guide to implement this architecture. We will create a custom RAG MCP server and connect it to your CLI.

👣 Step-by-Step Implementation Guide

Phase 1: Project and Environment Setup

1. **Create a Project Directory**: This will hold all your configuration files and code.

```
mkdir my_project_rag_mcp
cd my_project_rag_mcp
```

2. **Set Up a Python Virtual Environment**: This keeps dependencies organized.

```
python -m venv venv
# On macOS/Linux:
```

```
source venv/bin/activate
# On Windows:
.\venv\Scripts\activate
```

3. **Install Required Dependencies**: Install the necessary Python packages for building the MCP server and RAG pipeline.

```
pip install "mcp[cli]" langchain langchain-community chromadb
sentence-transformers
```

Phase 2: Setting Up the Vector Database and RAG

For this guide, we'll use **ChromaDB**, a lightweight vector database that runs locally and is simple to set up.

1. **Create the MCP Server Script**: Create a file named `rag_mcp_server.py` and use the following code as a template. This script does the heavy lifting: it loads your code, creates embeddings, and sets up the RAG tool.

```
# rag_mcp_server.py
from mcp.server.fastmcp import FastMCP
from langchain.chains import RetrievalQA
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import HuggingFaceEmbeddings

# Initialize the MCP server
mcp = FastMCP("Project-RAG")

# Set up the embedding model
embeddings = HuggingFaceEmbeddings(model_name="sentence-
transformers/all-MiniLM-L6-v2")

# Load your project's documents. You may need a more sophisticated
loader for multiple files.
# This is a basic example. For a real project, use a loader like
DirectoryLoader.
loader = TextLoader("/path/to/your/main/project/file.py")  # Update
this path
documents = loader.load()

# Split documents into chunks for processing
text_splitter = CharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
texts = text_splitter.split_documents(documents)

# Create and persist the vector store
vectorstore = Chroma.from_documents(documents=texts,
embedding=embeddings, persist_directory="./chroma_db")
```

```
retriever = vectorstore.as_retriever()

# Create the QA chain
from langchain.chat_models import ChatOpenAI
# Note: You can also configure this to use a local model or Qwen via
API
llm = ChatOpenAI(model="gpt-4") # You need an OpenAI API key for this
example
qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)

# Define the MCP tool that the AI CLI will call
@mcp.tool()
def query_project_code(question: str) -> str:
    """Use this tool to get answers about the project's codebase. Ask
specific questions about functions, classes, or logic."""
    result = qa_chain.invoke({"query": question})
    return result["result"]

if __name__ == "__main__":
    mcp.run()
```

Important Notes for the Script:

- **Document Loader**: The example uses `TextLoader` for a single file. For a full project, replace this with a `DirectoryLoader` from `langchain.document_loaders` to load all code files (e.g., `*.py`, `*.js`).
- **LLM for QA**: The example uses GPT-4 for answering questions. You can replace `ChatOpenAI` with other LangChain-supported LLMs, such as one pointing to Qwen's API.

Phase 3: Configuring Your AI CLIs

Now, you need to tell your Claude Code or Qwen Code CLI about your new MCP server. The configuration is typically done via a JSON settings file.

1. **Find Your CLI's Configuration Directory**:
   - **Claude Code**: The configuration is usually in `~/.claude/mcp.json` or within the Claude Desktop app settings.
   - **Qwen Code**: The configuration is typically in `~/.qwen/settings.json`.
2. **Edit the Configuration File**: Add a configuration for your RAG server. You'll need the full path to your Python executable and the `rag_mcp_server.py` script.

   Example configuration for Claude Code (`~/.claude/mcp.json`):

```
{
  "mcpServers": {
    "project-rag": {
      "command": "/path/to/your/venv/bin/python",
      "args": [
```

```
          "/full/path/to/your/my_project_rag_mcp/rag_mcp_server.py"
        ]
      }
    }
}
```

Example configuration for Qwen Code (`~/.qwen/settings.json`):

```
{
  "mcpServers": {
    "project-rag": {
      "command": "/path/to/your/venv/bin/python",
      "args": [
        "/full/path/to/your/my_project_rag_mcp/rag_mcp_server.py"
      ]
    }
  }
}
```

- **Windows Users**: The command path would look like `"C:\\path\\to\\your\\venv\\Scripts\\python.exe"`.

3. **Restart Your CLI**: Fully close and restart your Claude Code or Qwen Code application for the changes to take effect.

🚀 Using Your New RAG-Powered Assistant

After restarting, your AI assistant will have access to the new tool.

- **In Claude Code**: You can invoke the tool by typing `/project-rag` followed by your question.
- **In Qwen Code**: The tool should be available automatically. You can ask questions like, "Use the `query_project_code` tool to find all functions related to user authentication."

Example Queries:

- "How does the user authentication flow work?"
- "Where is the `calculate_total` function defined, and what other functions call it?"
- "What are the main components of the `User` class?"

💡 Tips for an "Impeccable" Setup

- **Better Document Processing**: For a large codebase, use a recursive directory loader and chunk your code intelligently to preserve context (e.g., don't split a function in the middle).
- **Leverage Qwen3 Models**: For potentially better performance, especially with code, consider using the `Qwen3-Embedding` model instead of the default one in the script. You would need to adjust the embedding setup in the server script.
- **Use a Managed Vector Database**: For very large codebases, a managed cloud solution like **Zilliz Cloud** (which runs Milvus) can offer better performance and scalability than ChromaDB.