Multi-Agent AI Coding Copilot – RAG Agent Design Document

Overview

This document outlines the architecture, tasks, and responsibilities of the **RAG Agent**, the first agent in a multi-agent reasoning system designed to solve context window limitations when building enterprise-grade AI coding copilots. This agent is responsible for retrieving relevant information from enterprise data using Retrieval-Augmented Generation (RAG), compressing it into an optimized prompt, and forwarding it to downstream agents for further processing.

🧠Agent-Based Multi-Step Reasoning – Workflow Explanation

Think of this system like a **smart factory**. A user gives a problem (like a machine that's not working), and each worker in the factory has a special job to solve the problem. These "workers" are called **agents**, and they all talk to each other to fix things faster and smarter.

😌Basic Flow

1. **User gives input** – This could be a question, a code file, or both.
2. **Agents work in a team** – Each agent does one job and passes their work to the next.
3. **System gives a final answer** – Neat and complete!

😾Agent Roles

1. 👎🏾RAG Agent – The Finder

Job: This agent's job is to **find helpful information** and **make it short and sweet** so the others can use it easily.

- Searches your codebase or documents using RAG (Retrieval-Augmented Generation).
- Uses tools like **ChromaDB** to find the most related files/chunks.
- Makes the results smaller using **prompt compression** (like squishing a big story into a short paragraph).
- Sends the compressed info to the next agent.

Handles These Scenarios:

- Just a question: Searches knowledge base.

- Just code: Figures out what the code might need.
- Both: Mixes both to improve search quality.

---

2. 👃🏾 Planner Agent – The Thinker

Job: This agent reads what the user (and RAG agent) gave and **breaks the task into smaller steps**.

- Like writing a to-do list.
- Example: "Fix memory bug" → Step 1: Check file, Step 2: Find leak, Step 3: Suggest fix
- Keeps things in order and hands them to the coder agent.

---

3. 🎉Coder Agent – The Coder

Job: This agent actually **writes or edits the code**.

- Uses tools like OpenAI, Claude, or CodeLlama to write based on the plan.
- Can understand code structures and make real code changes.
- Returns the updated code back to the team.

---

4. 😽Evaluator Agent – The Tester

Job: Checks the coder's work.

- Runs tests, scans for bugs, and makes sure the code works well.
- Can even give suggestions to improve code.
- If anything is wrong, it may send the code back to the coder to try again.

😇How These Agents Work Together

All the agents:

- **Talk using a shared memory** (like a notebook they all write in).
- Are connected through a **framework** like:
- **CrewAI** (clean and simple)
- **AutoGen** (super customizable)
- **LangGraph** (graph-style workflows)
- **LangChain Agents** (good if already using LangChain for RAG)

Each agent is a **Python function/class** that waits for input, does its job, and returns output.

🧠What Happens Based on User Input?

Input Type What Happens	
Only Query	RAG agent searches DB → Sends compressed info → Planner makes plan
Only File/Folder	RAG guesses intent (based on file structure) → Creates query → Proceeds
Just Code (No Query)	RAG infers what the user might want → Example: sees function → "Maybe refactor?"
Query + File	Best case! Combines query with file → More accurate search and planning

---

Framework Suggestions (Team Decision Point)

The team can choose from:

1. CrewAI

- Agent orchestration via Python classes
- Memory sharing, tool usage, and collaboration
- Flexible and lightweight

2. AutoGen (Microsoft)

- Supports function-calling agents
- Better for infrastructure-level control
- Async coordination and LLM-as-agent

3. LangGraph (LangChain)

- Visual graph-based agent interaction
- More intuitive debugging
- Best with LangChain ecosystem

4. LangChain Agents

- Ideal for workflows already using LangChain
- Integrates with Tools, Memory, and Retrieval

Recommendation: If you're already using ChromaDB and LangChain for RAG, then **LangChain Agents or LangGraph** is a suitable option for initial deployment.

RAG Agent (Your Assigned Agent)

👈🏽Primary Role

To fetch only the *most relevant information* needed to solve the user's query/code issue, and compress it into an LLM-friendly prompt using techniques like text-to-prompt compression.

👈🏿Tools Required

- **ChromaDB / FAISS / Weaviate** – Vector store for retrieval
- **LangChain Retriever** – Abstraction to search the vector store
- **LLM (e.g., GPT-4 or Claude)** – To perform prompt compression
- **LangChain Memory or External Storage** – Store retrieved histories

👇🏾Input Scenarios and Handling

The agent must intelligently handle:

1. Query-Only (No File)

- Use the user query to search relevant documents in ChromaDB
- Compress matching chunks into a prompt
- Forward to Planner agent

2. Only File or Whole Directory (No Query)

- Use metadata or recent memory to guess intent
- Extract file structure and top comments
- Summarize to form query-like instruction

3. Code Snippet Without Prompt

- Perform code classification: bug-fix? doc-gen? refactor?
- Infer context using code structure and variable names
- Retrieve related patterns or function usages

4. Query + File(s)

- Combine both inputs to enhance retrieval
- Rank results by hybrid retrieval (text + metadata)
- Deduplicate and compress before passing along

🧠Memory and Context Management

- Stores past queries and code context

- Useful for multi-turn interactions
- Works with LangChain's `ConversationBufferMemory`

😾Prompt Compression Techniques

Used to fit relevant data into LLM context window:

- Sentence Transformers + Cosine Similarity
- GPT-based summarization
- Graph compression (rank key relationships)
- Token ranking and filtering

👉🏻Output to Next Agent

- Final RAG-optimized compressed prompt
- Meta-data: source file, query type, tags
- Sent to Planner agent (or directly to Coder if simple task)

👉🏿Example Workflows

Query-Only:

> "Fix memory leak in C++ server"

- Searches vector DB → finds 3 relevant docs
- Compresses → sends compressed input to Planner

Code + No Query:

> Uploads `server.cpp`

- Extracts functions, header files
- Uses summarizer → detects memory problem → generates inferred query

---

👈🏼Summary

Your RAG Agent is the brain at the entry point. Its job is to reduce large inputs to their **most essential pieces of information** and fit them into the LLM's processing window — setting up all other agents for success. By using RAG and prompt compression, we address hallucinations, avoid context overflow, and give other agents only what they need to succeed.