ChatGPT

Functional Requirements for MCP-RAG Python Project

These functional requirements define a Python-based Model-Context Protocol (MCP) server that implements **retrieval-augmented generation (RAG)** and exposes management tools via the MCP interface. Requirements adhere to SMART principles (Specific, Measurable, Achievable, Relevant, Time-bound) and IEEE/NASA guidelines for completeness, correctness, consistency, and testability [1] [2].

1. Project and Packaging Standards

1. **Repository layout** – The project shall follow Python best practices: source code resides under a top-level `src/` directory; tests under `tests/`; documentation under `docs/`; and configuration scripts (e.g. `install.py`, CLI entry points) under `scripts/` or root. A `pyproject.toml` shall define metadata, dependencies and specify `requires-python = ">=3.10"`. A `README.md`, `CHANGELOG.md` and `LICENSE` shall be present. The repository shall use a `.gitignore` covering Python caches, build artefacts and user-specified ignore patterns.

2. **Supported Python versions** – The application shall support Python versions **3.10 through the latest stable release**. Python 3.10 is a hard minimum (MUST) because you explicitly require the codebase to run on version 3.10 or later. The code and dependencies must therefore work on Python 3.10 and the latest stable release at the time of deployment. Versions lower than 3.10 are not supported.

3. **Cross-platform support** – The code shall run on **Windows 10**, **Windows 11**, and **Ubuntu Linux 22.04 LTS or later**. Platform-specific paths and environment detection must be handled in a cross-platform manner. Any compiled dependencies must be supported on these platforms.

4. **Library compatibility checks** – Before using a third-party library, the installer or build process shall read the library's declared `python_requires` or classifiers to verify compatibility with Python 3.10+ and the latest Python release. If incompatible, the installer shall log a meaningful error and prevent installation.

5. **Version control and CI** – The repository shall include GitHub Actions workflows to run linting (e.g., `ruff` / `flake8`), unit tests with `pytest` and coverage measurement with `pytest-cov`, and to build documentation. Workflows must run on Ubuntu and Windows runners using **Python 3.10** and the latest Python version. Pull requests must pass all checks before merging.

2. Installation and Configuration

1. **Install script** – A Python script (`install.py`) shall create an isolated **virtual environment** in the project directory, detect the host operating system, install project dependencies from the lock file, and prompt the user (or read a supplied configuration file) for initial settings:
2. Path to the user's home directory (default is system's `$HOME` or `%USERPROFILE%`).
3. Default folder for storing MCP configuration data (`~/.mcp-agent-rag`).
4. Selection of the **embedding model** from available Ollama embedding models (see §5.1) and the **generative model** from available MIT/Apache-licensed models.
5. Default embedding chunk size and overlap (RAG parameters).

6. Active databases to load on server startup (list of names; may be empty).

7. **Configuration file** – The installer shall save the configuration to a JSON file under `~/.mcp-agent-rag/config.json`. Subsequent runs shall reuse this configuration unless overridden via command-line flags. The configuration shall include: list of defined databases with metadata (name, file path to FAISS index, description); embedding and generative model selections; and default RAG parameters.

8. **Command-line interface (CLI)** – The project shall expose a CLI (`mcp-rag`) installed via entry point in `pyproject.toml`. The CLI shall provide commands described in §4. CLI options shall allow specifying the configuration file (default `~/.mcp-agent-rag/config.json`).

3. Retrieval-Augmented Generation Pipeline

1. **Supported file formats** – The system shall ingest the following file types: plain text (`.txt`), Microsoft Office formats (`.docx`, `.xlsx`, `.pptx`), Open Document formats (`.odt`, `.ods`, `.odp`), PDFs (`.pdf`), and **software projects** (collections of source and build files). A software project is treated as a single document whose content comprises all files within the project directory except those excluded by `.gitignore` or `.svnignore`. Supported programming and scripting languages include C, C++, C#, Go, Rust, Java, Python, Windows batch scripts, PowerShell scripts, Bash scripts, ARM32 assembly, RISC-V assembly, and build files (Makefiles, CMake lists, compilation databases, map files, etc.).

2. **RAG preprocessing** – When adding documents to a database, the system shall:

3. **Extract text** from supported files using appropriate parsers.
4. **Clean** the text to remove extraneous characters while preserving headings, code blocks and other structural information.
5. **Chunk** the text into segments of configurable length with overlap, attach metadata (source file, page or chunk number, project name) and embed each chunk using the selected **embedding model**. These steps follow best-practice RAG pipelines [3].

6. **Index** the embeddings into a **FAISS** vector database along with metadata for retrieval [4].

7. **Embedding models** – The system shall use an Ollama embedding model (e.g., `nomic-embed-text` or `mxbai-embed-large`) to generate embeddings for documents and queries. The configuration must allow switching between available Ollama embedding models. Embedding functions shall call the Ollama `/api/embed` endpoint to produce vectors, as recommended by Ollama's RAG examples [5].

8. **Generative models** – To generate answers, the system shall use an Apache- or MIT-licensed LLM model capable of running on a single consumer GPU (e.g., NVIDIA RTX 4070) via Ollama. The default shall be **Mistral-7B-Instruct** (Apache-2.0) unless the configuration specifies another model. A separate embedding model will continue to be used for retrieval [6]. The system must verify that the selected model can run within the available GPU memory; otherwise it shall fall back to CPU with a warning.

9. **Database storage** – Each RAG database shall be stored as a FAISS index file along with metadata in the configuration file. Databases are identified by a **unique name** provided at creation. Databases shall be stored under `~/.mcp-agent-rag/databases/<database_name>/` by default. The index may include additional persistent state (e.g., pickled metadata).

4. MCP Server and Tools (AGNO-based)

The server shall be implemented using the **Agno** framework because it provides built-in MCP support and agentic RAG features [7]. It must implement the following MCP tools; each tool description must be clear so that an agentic AI client understands its purpose.

1. `database/create` **tool** – Creates a new database with a **unique name**. Inputs: desired name (string). If a database with the same name already exists, the tool shall return an error. Output: confirmation message with database name and storage location.

2. `database/add` **tool** – Adds documents to an existing database. Inputs:

3. `database_name` (string) – name of the target database.
4. One of: (a) `path` (string) – absolute or relative path to a file or directory; (b) `glob` (string) – path with wildcard; (c) `url` (string) – HTTP/HTTPS URL to a document or web page. An optional flag `recursive` (boolean) indicates that subdirectories shall be traversed.
5. Additional flags: `skip_existing` (boolean) to avoid re-ingesting files already in the database.

Behaviour: - Validate that the database exists; if not, return an error. - Determine all matching files (respecting `.gitignore` / `.svnignore` when ingesting projects) and estimate the count. Display the total count to the user. - Process each file in sequence; before processing each file, print progress (e.g., “Processing 3 of 12: …”). If the user presses `Ctrl+K`, skip the current file and continue with the next one (this key combination works on Windows and Unix shells). If parsing fails for a file, log a meaningful error and continue. - Perform RAG preprocessing (extract, clean, chunk, embed, index) and update the database. - At completion, output a summary (number of files processed, skipped, failed).

1. `database/list` **tool** – Returns a list of all databases known to the server. For each database, it shall include the name, description (user-supplied or auto-generated summarising the data), number of documents/chunks, and last updated timestamp. This tool allows an agent to choose which databases to activate.

2. `query/get_data` **tool** – Retrieves context for a user’s prompt from the active databases. Inputs: `prompt` (string), `max_results` (integer, default 5). Behaviour:

3. The tool uses an **agentic agent** built with Agno to determine which databases are relevant to the prompt. The agent generates sub-queries tailored to each active database, performs vector similarity search to retrieve top `k` chunks from each, and may optionally re-rank the results.

4. It then composes a **succinct, non-redundant** combined context string by aggregating and deduplicating retrieved chunks. The response must be formatted in plain text so that it can be easily inserted into subsequent prompts. The output shall include citations (e.g., file names and chunk identifiers) so that traceability is maintained.

5. `server/start` **command** (CLI, not an MCP tool) – Starts the MCP server. Inputs:

6. `--active-databases` – comma-separated list of database names to load into memory; required. These active databases are the only ones accessible to `query/get_data`. If a listed database does not exist, the command shall exit with an error.
7. `--config` – path to configuration file; default `~/.mcp-agent-rag/config.json`.
8. `--transport` – one of `stdio` (default) or `http`; if `http`, optional `--host` (default `127.0.0.1`) and `--port` (default `8080`) may be provided.

Behaviour: - Load the configuration file and verify that the specified active databases exist and load their FAISS indices into memory. - Expose MCP endpoints via the selected transport. For HTTP, implement server-sent events for streaming responses as described in the MCP specification [8]. Validate the `Origin` header for security and require local connections unless explicitly configured [9]. - Log startup information: active databases, transport, host and port.

1. **Error handling** – All tools must return meaningful error messages and never expose stack traces. For example, if a user tries to create a database with a reserved name, the response shall state "Database 'X' already exists"; if document ingestion fails, it shall state the reason and continue with remaining files. Error responses must be valid MCP error objects following JSON-RPC conventions [10].

5. Agentic Agent Implementation

1. **Framework** – Implement the agentic agent using the **Agno** framework's built-in agentic RAG capabilities. Agno provides durable execution, memory, human-in-the-loop workflows and first-class MCP support [7]. The agent shall manage conversation state across requests and incorporate session history when constructing sub-queries.

2. **Database awareness** – The agent shall maintain a list of active databases loaded at server startup. For each `query/get_data` request, it shall decide which databases may contain relevant information, generate sub-queries accordingly and combine results. The decision may be based on vector similarity between the prompt and pre-computed database summaries or heuristics.

3. **Aggregation logic** – After retrieving top chunks from each selected database, the agent shall deduplicate overlapping information, rank the chunks by relevance, and compose a combined context string. The combined context must avoid ambiguities and duplicate facts; sections should be clearly delineated (e.g., separate paragraphs for each source). The output shall not exceed a configurable character limit (default 4 000 characters) to ensure it fits into subsequent prompts.

4. **Extensibility** – The design shall allow adding other AGNO tools or custom vector stores in the future. Each tool shall be implemented as a separate function class to enable registration with Agno's MCP runtime.

6. Logging and Metrics

1. **Progress reporting** – For document ingestion, the CLI shall report the total number of files to process and display incremental progress. Each processed file shall be logged with status (processed, skipped, error). The final summary shall include counts of processed, skipped and failed files.

2. **Error logs** – Errors shall be logged to both the console and a log file under `~/.mcp-agent-rag/logs/` with timestamp and severity level. The log file shall rotate when exceeding 10 MB.

3. **Test coverage reporting** – The test suite shall generate coverage reports in both terminal and XML formats (`coverage.xml`). The CI pipeline shall fail if per-module or overall coverage drops below 90 %.

7. Constraints and Time-bound Requirements

1. **Completion deadlines** – The project shall have a release candidate within **6 weeks** of project initiation. An initial prototype (functional ingestion and basic query) must be ready within **3 weeks**. These deadlines are set to ensure timely delivery and allow iterative improvements.

2. **Modifiability and traceability** – Requirements shall be traceable to code, tests and documentation. Changes in functionality must update the requirements, documentation and tests accordingly, consistent with IEEE guidelines [1]. Each requirement shall have a unique identifier (e.g., FR-1, FR-2) for traceability; these identifiers are implicitly provided by the numbering in this document.

8. Non-functional considerations

Although primarily functional, the project must adhere to non-functional guidelines mentioned by NASA and IEEE (clarity, completeness, consistency, traceability and verifiability). All requirements must be testable [2]. The system shall respect user privacy and ensure that no data is transmitted without explicit user consent [11].

---

[1] What is a Requirement | Cardinal
https://cardinal.cels.anl.gov/sqa/what_is_a_requirement.html

[2] Appendix C: How to Write a Good Requirement - NASA
https://www.nasa.gov/reference/appendix-c-how-to-write-a-good-requirement/

[3] The Role of Data Preprocessing in RAG | deepset Blog
https://www.deepset.ai/blog/preprocessing-rag

[4] What is Retrieval Augmented Generation (RAG)? | Databricks
https://www.databricks.com/glossary/retrieval-augmented-generation-rag

[5] [6] Embedding models · Ollama Blog
https://ollama.com/blog/embedding-models

[7] README.md
https://github.com/biodoia/ango/blob/main/README.md

8 9 Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-11-25/basic/transports

10 Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-11-25/basic

11 Specification - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-11-25