ChatGPT

Acceptance Test Specification for MCP-RAG Python Project

This document defines acceptance tests verifying that the functional requirements of the MCP-RAG project have been met. Tests follow a Given-When-Then format [1] and are grouped by feature. Each test case includes preconditions, actions and expected outcomes. All tests shall be automated using `pytest` and `pytest-cov`. Coverage metrics per module and overall must exceed 90 %.

1. Installation and Configuration

Test 1.1 – Install script creates virtual environment and installs dependencies

Preconditions: - A clean system with **Python 3.10** installed. - The project repository has been cloned. - No existing virtual environment in the project directory.

Steps: 1. Run `python install.py` with default options.

Expected: - A virtual environment is created under `.venv` (or configured location). - Dependencies are installed without errors. - A configuration file is written to `~/.mcp-agent-rag/config.json` with default values. - The script prompts for model selection and records the selection.

Test 1.2 – Install script reads configuration from file

Preconditions: - A configuration file exists at `~/.mcp-agent-rag/config.json` with valid settings.

Steps: 1. Run `python install.py --config ~/.mcp-agent-rag/config.json`.

Expected: - The installer reads the configuration file without prompting for settings. - The virtual environment is created and dependencies installed as specified.

2. Database Management Tools

Test 2.1 – Create a new database

Preconditions: - MCP server is not running. - A configuration file exists. - No database named `testdb` exists.

Steps: 1. Run `mcp-rag database create --name testdb`.

Expected: - A new FAISS index directory is created under `~/.mcp-agent-rag/databases/testdb`. - The configuration file lists `testdb` with an empty description. - The CLI outputs a success message including the database name and path.

Test 2.2 – Prevent duplicate database names

Preconditions: - A database named `testdb` already exists in configuration.

Steps: 1. Run `mcp-rag database create --name testdb`.

Expected: - The command fails with an error message: “Database ‘testdb’ already exists.” - No new directory is created.

Test 2.3 – List databases

Preconditions: - Configuration lists at least two databases (`db1`, `db2`) with descriptions.

Steps: 1. Run `mcp-rag database list`.

Expected: - The CLI prints a table with each database name, description, document count and last updated timestamp. - The number of listed databases matches the configuration.

Test 2.4 – Add documents via path and wildcard

Preconditions: - Database `testdb` exists. - A directory `sample_docs/` contains files `a.txt`, `b.pdf`, and `ignore.tmp`.

Steps: 1. Run `mcp-rag database add --database testdb --path sample_docs/*.txt`.

Expected: - Only `a.txt` is ingested; `b.pdf` is ignored because of the wildcard. - The CLI displays “1 files to process” at start. - Progress messages show each file processed. - The FAISS index for `testdb` contains embeddings for `a.txt`.

Test 2.5 – Add documents recursively with skip key

Preconditions: - Database `testdb` exists. - A directory tree `project/` contains multiple nested files including `main.py` and `README.md`. `.gitignore` lists `*.pyc`.

Steps: 1. Run `mcp-rag database add --database testdb --path project/ --recursive`. 2. When processing the second file, press **Ctrl+K**.

Expected: - The CLI determines the total file count excluding `.pyc` files (per `.gitignore`). - After pressing Ctrl+K, the current file is skipped and the process continues with the next file. - A final summary reports the number of processed and skipped files.

Test 2.6 – Add documents from URL

Preconditions: - Database `testdb` exists. - Internet connectivity is available.

Steps: 1. Run `mcp-rag database add --database testdb --url https://example.com/page.html`.

Expected: - The page is downloaded and converted to text. - Text is cleaned, chunked, embedded and added to the index. - A success message summarises the ingestion.

Test 2.7 – Handle unsupported file format gracefully

Preconditions: - Database `testdb` exists. - A file `archive.zip` is present.

Steps: 1. Run `mcp-rag database add --database testdb --path archive.zip`.

Expected: - The CLI reports that `.zip` is unsupported and skips the file. - No exception is thrown.

3. Server Startup and Transports

Test 3.1 – Start server with active databases via stdio

Preconditions: - Databases `db1` and `db2` exist and have indexes.

Steps: 1. Run `mcp-rag server start --active-databases db1,db2`.

Expected: - The server loads `db1` and `db2` indexes into memory. - It logs the transport (`stdio`) and active databases. - The process waits for JSON-RPC input on stdin without exiting.

Test 3.2 – Start server with HTTP transport

Preconditions: - Database `db1` exists.

Steps: 1. Run `mcp-rag server start --active-databases db1 --transport http --host 127.0.0.1 --port 9000`. 2. Send an HTTP POST request to `http://127.0.0.1:9000` with a JSON-RPC `resources/list` request.

Expected: - The server starts and listens on port 9000. - The HTTP response includes a valid JSON-RPC result listing resources. - The `Origin` header must be validated; requests from disallowed origins return an error [2].

Test 3.3 – Reject missing active databases

Preconditions: - Only database `db1` exists.

Steps: 1. Run `mcp-rag server start --active-databases db1,db2`.

Expected: - The server fails to start and prints an error: “Database ‘db2’ does not exist.”

4. Querying for Data

Test 4.1 – Retrieve context for prompt

Preconditions: - Server is running with database `db1` active. - `db1` contains documents with embedded chunks.

Steps: 1. Send a JSON-RPC `query/get_data` request with a prompt “What is the purpose of file X?”.

Expected: - The agent inspects available databases and selects `db1`. - It generates sub-queries, retrieves relevant chunks and composes a response. - The response is plain text containing the requested information without duplicates and includes citations to source files.

Test 4.2 – Retrieve context from multiple databases

Preconditions: - Server runs with `db1` and `db2` active. - Both databases contain relevant information about “encryption” and “networking”.

Steps: 1. Send a `query/get_data` request with the prompt “Explain how the encryption module interacts with networking”.

Expected: - The agent decides to query both `db1` and `db2`. - Retrieved chunks from both databases are merged, deduplicated and summarised. - The response text fits within the configured character limit and contains citations from both databases.

Test 4.3 – Validate max_results parameter

Preconditions: - Server is running with database `db1` active.

Steps: 1. Send `query/get_data` with `prompt="..."` and `max_results=2`.

Expected: - The agent returns at most two chunks per database. - The returned context includes at most two citations.

5. Error Handling and Logging

Test 5.1 – Meaningful error on nonexistent database

Preconditions: - Server is running with `db1` active.

Steps: 1. Send `database/add` request targeting non-existent `dbX`.

Expected: - The response is a JSON-RPC error object stating that `dbX` does not exist. - No action is taken on existing databases.

Test 5.2 – Log file rotation

Preconditions: - Logging is configured with a 10 MB file limit.

Steps: 1. Generate log entries (e.g., by adding many documents) until the log file exceeds 10 MB.

Expected: - The logger rotates the log file: a new log file is created and logging continues without interruption.

6. Test Coverage

Test 6.1 – Coverage threshold

Procedure: 1. Run `pytest --cov=src --cov-report=xml` on the repository.

Expected: - The per-module and overall coverage metrics are ≥90 %. If coverage falls below this threshold, the test fails.

7. Non-functional Tests

Test 7.1 – Privacy and consent

Procedure: 1. Review network communications when adding documents and serving queries.

Expected: - No data (document contents or embeddings) is transmitted to external servers unless the configuration explicitly specifies remote services. All network requests respect user consent and configuration [3].

Test 7.2 – Model compatibility check

Procedure: 1. Attempt to install a dependency that declares `python_requires < 3.10` via the install script.

Expected: - The installer detects the incompatibility, aborts installation and logs an error describing the issue.

Test 7.3 – OS detection

Procedure: 1. Run the install script on Windows 10, Windows 11 and Ubuntu 22.04.

Expected: - The script correctly identifies the operating system. - The virtual environment is created and dependencies installed on each platform.

---

[1] A guide to acceptance testing
https://qase.io/blog/acceptance-testing/

[2] Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-11-25/basic/transports

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