



Web Crawling and RAG Capabilities for AI Agents and AI Coding Assistants

MIT license

☆ 1k stars 354 forks 39 watching Branches Activity

Tags

Public repository

1 Branch 0 Tags

Code

...

coleam00 Documentation updates for knowledge graphs a1c16f6 · 4 days ago

	knowledge_graphs	Knowledge graph functionality add...	last week
	src	Knowledge graph functionality add...	last week
	.dockerignore	Crawl4AI RAG MCP Server Initial C...	last month
	.env.example	Documentation updates for knowle...	4 days ago
	.gitattributes	Initial commit	last month
	.gitignore	Knowledge graph functionality add...	last week
	Dockerfile	Fixed Dockerfile command	3 weeks ago
	LICENSE	Initial commit	last month
	README.md	Documentation updates for knowle...	4 days ago
	crawled_pages.sql	Added various RAG strategies and ...	3 weeks ago
	pyproject.toml	Knowledge graph validation v1	last week
	uv.lock	Knowledge graph validation v1	last week

README MIT license

Crawl4AI RAG MCP Server

Web Crawling and RAG Capabilities for AI Agents and AI Coding Assistants

A powerful implementation of the [Model Context Protocol \(MCP\)](#) integrated with [Crawl4AI](#) and [Supabase](#) for providing AI agents and AI coding assistants with advanced web crawling and RAG capabilities.

With this MCP server, you can **scrape anything** and then **use that knowledge anywhere** for RAG.

The primary goal is to bring this MCP server into [Archon](#) as I evolve it to be more of a knowledge engine for AI coding assistants to build AI agents. This first version of the Crawl4AI/RAG MCP server will be improved upon greatly soon, especially making it more configurable so you can use different embedding models and run everything locally with Ollama.

Consider this GitHub repository a testbed, hence why I haven't been super actively address issues and pull requests yet. I certainly will though as I bring this into Archon V2!

Overview

This MCP server provides tools that enable AI agents to crawl websites, store content in a vector database (Supabase), and perform RAG over the crawled content. It follows the best practices for building MCP servers based on the [Mem0 MCP server template](#) I provided on my channel previously.

The server includes several advanced RAG strategies that can be enabled to enhance retrieval quality:

- **Contextual Embeddings** for enriched semantic understanding
- **Hybrid Search** combining vector and keyword search
- **Agentic RAG** for specialized code example extraction
- **Reranking** for improved result relevance using cross-encoder models
- **Knowledge Graph** for AI hallucination detection and repository code analysis

See the [Configuration section](#) below for details on how to enable and configure these strategies.

Vision

The Crawl4AI RAG MCP server is just the beginning. Here's where we're headed:

1. **Integration with Archon:** Building this system directly into [Archon](#) to create a comprehensive knowledge engine for AI coding assistants to build better AI agents.
2. **Multiple Embedding Models:** Expanding beyond OpenAI to support a variety of embedding models, including the ability to run everything locally with Ollama for complete control and privacy.
3. **Advanced RAG Strategies:** Implementing sophisticated retrieval techniques like contextual retrieval, late chunking, and others to move beyond basic "naive lookups" and significantly enhance the power and precision of the RAG system, especially as it integrates with Archon.
4. **Enhanced Chunking Strategy:** Implementing a Context 7-inspired chunking approach that focuses on examples and creates distinct, semantically meaningful sections for each chunk, improving retrieval precision.
5. **Performance Optimization:** Increasing crawling and indexing speed to make it more realistic to "quickly" index new documentation to then leverage it within the same prompt in an AI coding assistant.

Features

- **Smart URL Detection:** Automatically detects and handles different URL types (regular webpages, sitemaps, text files)
- **Recursive Crawling:** Follows internal links to discover content
- **Parallel Processing:** Efficiently crawls multiple pages simultaneously
- **Content Chunking:** Intelligently splits content by headers and size for better processing
- **Vector Search:** Performs RAG over crawled content, optionally filtering by data source for precision
- **Source Retrieval:** Retrieve sources available for filtering to guide the RAG process

Tools

The server provides essential web crawling and search tools:

Core Tools (Always Available)

1. **crawl_single_page** : Quickly crawl a single web page and store its content in the vector database
2. **smart_crawl_url** : Intelligently crawl a full website based on the type of URL provided (sitemap, llms-full.txt, or a regular webpage that needs to be crawled recursively)
3. **get_available_sources** : Get a list of all available sources (domains) in the database
4. **perform_rag_query** : Search for relevant content using semantic search with optional source filtering

Conditional Tools

5. **search_code_examples** (requires `USE_AGENTIC_RAG=true`): Search specifically for code examples and their summaries from crawled documentation. This tool provides targeted code snippet retrieval for AI coding assistants.

Knowledge Graph Tools (requires `USE_KNOWLEDGE_GRAPH=true`, see below)

6. **parse_github_repository** : Parse a GitHub repository into a Neo4j knowledge graph, extracting classes, methods, functions, and their relationships for hallucination detection
7. **check_ai_script_hallucinations** : Analyze Python scripts for AI hallucinations by validating imports, method calls, and class usage against the knowledge graph
8. **query_knowledge_graph** : Explore and query the Neo4j knowledge graph with commands like `repos`, `classes`, `methods`, and custom Cypher queries

Prerequisites

- [Docker/Docker Desktop](#) if running the MCP server as a container (recommended)
- [Python 3.12+](#) if running the MCP server directly through uv
- [Supabase](#) (database for RAG)
- [OpenAI API key](#) (for generating embeddings)
- [Neo4j](#) (optional, for knowledge graph functionality) - see [Knowledge Graph Setup](#) section

Installation

Using Docker (Recommended)

1. Clone this repository:

```
git clone https://github.com/coleam00/mcp-crawl4ai-rag.git
cd mcp-crawl4ai-rag
```



2. Build the Docker image:

```
docker build -t mcp/crawl4ai-rag --build-arg PORT=8051 .
```



3. Create a `.env` file based on the configuration section below

Using uv directly (no Docker)

1. Clone this repository:

```
git clone https://github.com/coleam00/mcp-crawl4ai-rag.git
cd mcp-crawl4ai-rag
```



2. Install uv if you don't have it:

```
pip install uv
```



3. Create and activate a virtual environment:

```
uv venv
.venv\Scripts\activate
# on Mac/Linux: source .venv/bin/activate
```



4. Install dependencies:

```
uv pip install -e .
crawl4ai-setup
```



5. Create a `.env` file based on the configuration section below

Database Setup

Before running the server, you need to set up the database with the pgvector extension:

1. Go to the SQL Editor in your Supabase dashboard (create a new project first if necessary)
2. Create a new query and paste the contents of `crawled_pages.sql`

3. Run the query to create the necessary tables and functions

Knowledge Graph Setup (Optional)

To enable AI hallucination detection and repository analysis features, you need to set up Neo4j.

Also, the knowledge graph implementation isn't fully compatible with Docker yet, so I would recommend right now running directly through uv if you want to use the hallucination detection within the MCP server!

For installing Neo4j:

Local AI Package (Recommended)

The easiest way to get Neo4j running locally is with the [Local AI Package](#) - a curated collection of local AI services including Neo4j:

1. Clone the Local AI Package:

```
git clone https://github.com/coleam00/local-ai-packaged.git
cd local-ai-packaged
```



2. **Start Neo4j:** Follow the instructions in the Local AI Package repository to start Neo4j with Docker Compose

3. **Default connection details:**

- URI: bolt://localhost:7687
- Username: neo4j
- Password: Check the Local AI Package documentation for the default password

Manual Neo4j Installation

Alternatively, install Neo4j directly:

1. **Install Neo4j Desktop:** Download from neo4j.com/download

2. **Create a new database:**

- Open Neo4j Desktop
- Create a new project and database
- Set a password for the neo4j user
- Start the database

3. **Note your connection details:**

- URI: bolt://localhost:7687 (default)
- Username: neo4j (default)
- Password: Whatever you set during creation

Configuration

Create a `.env` file in the project root with the following variables:

```
# MCP Server Configuration
HOST=0.0.0.0
PORT=8051
TRANSPORT=sse

# OpenAI API Configuration
OPENAI_API_KEY=your_openai_api_key

# LLM for summaries and contextual embeddings
MODEL_CHOICE=gpt-4.1-nano

# RAG Strategies (set to "true" or "false", default to "false")
USE_CONTEXTUAL_EMBEDDINGS=false
USE_HYBRID_SEARCH=false
USE_AGENTIC_RAG=false
USE_RERANKING=false
USE_KNOWLEDGE_GRAPH=false

# Supabase Configuration
SUPABASE_URL=your_supabase_project_url
SUPABASE_SERVICE_KEY=your_supabase_service_key

# Neo4j Configuration (required for knowledge graph functionality)
NEO4J_URI=bolt://localhost:7687
NEO4J_USER=neo4j
NEO4J_PASSWORD=your_neo4j_password
```



RAG Strategy Options

The Crawl4AI RAG MCP server supports four powerful RAG strategies that can be enabled independently:

1. USE_CONTEXTUAL_EMBEDDINGS

When enabled, this strategy enhances each chunk's embedding with additional context from the entire document. The system passes both the full document and the specific chunk to an LLM (configured via `MODEL_CHOICE`) to generate enriched context that gets embedded alongside the chunk content.

- **When to use:** Enable this when you need high-precision retrieval where context matters, such as technical documentation where terms might have different meanings in different sections.
- **Trade-offs:** Slower indexing due to LLM calls for each chunk, but significantly better retrieval accuracy.
- **Cost:** Additional LLM API calls during indexing.

2. USE_HYBRID_SEARCH

Combines traditional keyword search with semantic vector search to provide more comprehensive results. The system performs both searches in parallel and intelligently merges results, prioritizing documents that appear in both result sets.

- **When to use:** Enable this when users might search using specific technical terms, function names, or when exact keyword matches are important alongside semantic understanding.

- **Trade-offs:** Slightly slower search queries but more robust results, especially for technical content.
- **Cost:** No additional API costs, just computational overhead.

3. USE_AGENTIC_RAG

Enables specialized code example extraction and storage. When crawling documentation, the system identifies code blocks (≥ 300 characters), extracts them with surrounding context, generates summaries, and stores them in a separate vector database table specifically designed for code search.

- **When to use:** Essential for AI coding assistants that need to find specific code examples, implementation patterns, or usage examples from documentation.
- **Trade-offs:** Significantly slower crawling due to code extraction and summarization, requires more storage space.
- **Cost:** Additional LLM API calls for summarizing each code example.
- **Benefits:** Provides a dedicated `search_code_examples` tool that AI agents can use to find specific code implementations.

4. USE_RERANKING

Applies cross-encoder reranking to search results after initial retrieval. Uses a lightweight cross-encoder model (`cross-encoder/ms-marco-MiniLM-L-6-v2`) to score each result against the original query, then reorders results by relevance.

- **When to use:** Enable this when search precision is critical and you need the most relevant results at the top. Particularly useful for complex queries where semantic similarity alone might not capture query intent.
- **Trade-offs:** Adds ~100-200ms to search queries depending on result count, but significantly improves result ordering.
- **Cost:** No additional API costs - uses a local model that runs on CPU.
- **Benefits:** Better result relevance, especially for complex queries. Works with both regular RAG search and code example search.

5. USE_KNOWLEDGE_GRAPH

Enables AI hallucination detection and repository analysis using Neo4j knowledge graphs. When enabled, the system can parse GitHub repositories into a graph database and validate AI-generated code against real repository structures. (NOT fully compatible with Docker yet, I'd recommend running through uv)

- **When to use:** Enable this for AI coding assistants that need to validate generated code against real implementations, or when you want to detect when AI models hallucinate non-existent methods, classes, or incorrect usage patterns.
- **Trade-offs:** Requires Neo4j setup and additional dependencies. Repository parsing can be slow for large codebases, and validation requires repositories to be pre-indexed.
- **Cost:** No additional API costs for validation, but requires Neo4j infrastructure (can use free local installation or cloud AuraDB).
- **Benefits:** Provides three powerful tools: `parse_github_repository` for indexing codebases, `check_ai_script_hallucinations` for validating AI-generated code, and `query_knowledge_graph` for exploring indexed repositories.

You can now tell the AI coding assistant to add a Python GitHub repository to the knowledge graph like:

"Add <https://github.com/pydantic/pydantic-ai.git> to the knowledge graph"

Make sure the repo URL ends with .git.

You can also have the AI coding assistant check for hallucinations with scripts it just created, or you can manually run the command:

```
python knowledge_graphs/ai_hallucination_detector.py [full path to your script to analyze]
```



Recommended Configurations

For general documentation RAG:

```
USE_CONTEXTUAL_EMBEDDINGS=false  
USE_HYBRID_SEARCH=true  
USE_AGENTIC_RAG=false  
USE_RERANKING=true
```



For AI coding assistant with code examples:

```
USE_CONTEXTUAL_EMBEDDINGS=true  
USE_HYBRID_SEARCH=true  
USE_AGENTIC_RAG=true  
USE_RERANKING=true  
USE_KNOWLEDGE_GRAPH=false
```



For AI coding assistant with hallucination detection:

```
USE_CONTEXTUAL_EMBEDDINGS=true  
USE_HYBRID_SEARCH=true  
USE_AGENTIC_RAG=true  
USE_RERANKING=true  
USE_KNOWLEDGE_GRAPH=true
```



For fast, basic RAG:

```
USE_CONTEXTUAL_EMBEDDINGS=false  
USE_HYBRID_SEARCH=true  
USE_AGENTIC_RAG=false  
USE_RERANKING=false  
USE_KNOWLEDGE_GRAPH=false
```



Running the Server

Using Docker

```
docker run --env-file .env -p 8051:8051 mcp/crawl4ai-rag
```



Using Python

```
uv run src/crawl4ai_mcp.py
```



The server will start and listen on the configured host and port.

Integration with MCP Clients

SSE Configuration

Once you have the server running with SSE transport, you can connect to it using this configuration:

```
{
  "mcpServers": {
    "crawl4ai-rag": {
      "transport": "sse",
      "url": "http://localhost:8051/sse"
    }
  }
}
```



Note for Windsurf users: Use `serverUrl` instead of `url` in your configuration:

```
{
  "mcpServers": {
    "crawl4ai-rag": {
      "transport": "sse",
      "serverUrl": "http://localhost:8051/sse"
    }
  }
}
```



Note for Docker users: Use `host.docker.internal` instead of `localhost` if your client is running in a different container. This will apply if you are using this MCP server within n8n!

Note for Claude Code users:

```
claude mcp add-json crawl4ai-rag '{"type":"http","url":"http://localhost:8051/sse"}' --
scope user
```



Stdio Configuration

Add this server to your MCP configuration for Claude Desktop, Windsurf, or any other MCP client:

```
{
  "mcpServers": {
    "crawl4ai-rag": {
      "command": "python",
      "args": ["path/to/crawl4ai-mcp/src/crawl4ai_mcp.py"],
      "env": {
```



```

    "TRANSPORT": "stdio",
    "OPENAI_API_KEY": "your_openai_api_key",
    "SUPABASE_URL": "your_supabase_url",
    "SUPABASE_SERVICE_KEY": "your_supabase_service_key",
    "USE_KNOWLEDGE_GRAPH": "false",
    "NEO4J_URI": "bolt://localhost:7687",
    "NEO4J_USER": "neo4j",
    "NEO4J_PASSWORD": "your_neo4j_password"
  }
}
}

```

Docker with Stdio Configuration

```

{
  "mcpServers": {
    "crawl4ai-rag": {
      "command": "docker",
      "args": ["run", "--rm", "-i",
        "-e", "TRANSPORT",
        "-e", "OPENAI_API_KEY",
        "-e", "SUPABASE_URL",
        "-e", "SUPABASE_SERVICE_KEY",
        "-e", "USE_KNOWLEDGE_GRAPH",
        "-e", "NEO4J_URI",
        "-e", "NEO4J_USER",
        "-e", "NEO4J_PASSWORD",
        "mcp/crawl4ai"],
      "env": {
        "TRANSPORT": "stdio",
        "OPENAI_API_KEY": "your_openai_api_key",
        "SUPABASE_URL": "your_supabase_url",
        "SUPABASE_SERVICE_KEY": "your_supabase_service_key",
        "USE_KNOWLEDGE_GRAPH": "false",
        "NEO4J_URI": "bolt://localhost:7687",
        "NEO4J_USER": "neo4j",
        "NEO4J_PASSWORD": "your_neo4j_password"
      }
    }
  }
}

```

Knowledge Graph Architecture

The knowledge graph system stores repository code structure in Neo4j with the following components:

Core Components (`knowledge_graphs/` folder):

- **`parse_repo_into_neo4j.py`** : Clones and analyzes GitHub repositories, extracting Python classes, methods, functions, and imports into Neo4j nodes and relationships
- **`ai_script_analyzer.py`** : Parses Python scripts using AST to extract imports, class instantiations, method calls, and function usage

- **knowledge_graph_validator.py** : Validates AI-generated code against the knowledge graph to detect hallucinations (non-existent methods, incorrect parameters, etc.)
- **hallucination_reporter.py** : Generates comprehensive reports about detected hallucinations with confidence scores and recommendations
- **query_knowledge_graph.py** : Interactive CLI tool for exploring the knowledge graph (functionality now integrated into MCP tools)

Knowledge Graph Schema:

The Neo4j database stores code structure as:

Nodes:

- **Repository** : GitHub repositories
- **File** : Python files within repositories
- **Class** : Python classes with methods and attributes
- **Method** : Class methods with parameter information
- **Function** : Standalone functions
- **Attribute** : Class attributes

Relationships:

- **Repository** -[:CONTAINS]-> **File**
- **File** -[:DEFINES]-> **Class**
- **File** -[:DEFINES]-> **Function**
- **Class** -[:HAS_METHOD]-> **Method**
- **Class** -[:HAS_ATTRIBUTE]-> **Attribute**

Workflow:

1. **Repository Parsing**: Use `parse_github_repository` tool to clone and analyze open-source repositories
2. **Code Validation**: Use `check_ai_script_hallucinations` tool to validate AI-generated Python scripts
3. **Knowledge Exploration**: Use `query_knowledge_graph` tool to explore available repositories, classes, and methods

Building Your Own Server

This implementation provides a foundation for building more complex MCP servers with web crawling

Releases




No releases published



Packages

No packages published

Contributors 3

-  coleam00 Cole Medin
-  ajaygunalan Ajay Gunalan
-  ye-chuan

Languages

- Python 98.1%
- PLpgSQL 1.8%
- Dockerfile 0.1%

