Microsoft's GraphRAG + AutoGen + Ollama + Chainlit = Local & Free Multi-Agent RAG Superbot

Guide to constructing an agentic-GraphRAG retrieval application—all codes in my GitHub repo.

Karthik Rajan, Ph.D · Follow

Published in AI Advances · 11 min read · Jul 15, 2024

Graphical abstract of the integration and key components

Retrieval-augmented generation (RAG) is a powerful tool that equips large language models (LLMs) with the ability to access real-world data for more informed responses. This is achieved by integrating the models with a vector database for real-time learning and adaptation. This feature makes RAG a preferred choice for applications such as chatbots and virtual assistants, where the demand for accurate and sensible responses in real time is high. An advanced variant of this, known as Graph Retrieval-Augmented Generation (GraphRAG), merges the benefits of graph-based knowledge retrieval with LLMs, further enhancing the capabilities in natural language processing. Unlike traditional RAG methods that rely on vector similarity searches, GraphRAG constructs a structured knowledge graph from raw text,

capturing entities, relationships, and critical claims. This can enhance LLMs' ability to understand and synthesize complex datasets and their relationships, yielding more accurate and contextually grounded responses.

Extracted from a paper by Markus Beuhler from MIT (link here)

AutoGen is a tool by Microsoft that streamlines the development of intricate applications based on multi-agent LLMs by automating and optimizing workflows that were once complicated and required significant manual effort. Picture AutoGen as a platform where you can interact with multiple GPTs instead of just one. Each GPT acts as an individual "agent", playing a unique part in a comprehensive operation. Combining GraphRAG's retrieval strengths with AutoGen AI agents' conversational and task-oriented functionalities results in robust AI assistants capable of efficiently handling detailed queries, generating and executing codes, creating multi-page scientific reports, and conducting data analysis. Furthermore, offline local LLMs, such as those from Ollama or LM Studio, ensure cost-effective and secure data processing. Local LLMs eliminate the high costs and privacy risks associated with online LLMs, keeping sensitive data within the organization and reducing operational expenses.

> *This article will guide you on constructing a multi-agent AI application with GraphRAG retrieval system, which operates entirely on your local*

> *machine and is available at no charge. Here are the key components of this application:*

1. GraphRAG's knowledge search methods are integrated with an AutoGen agent via function calling.
2. GraphRAG (local & global search) is configured to support local models from Ollama for inference and embedding.
3. AutoGen was extended to support function calling with non-OpenAI LLMs from Ollama via the Lite-LLM proxy server.
4. Chainlit UI to handle continuous conversations, multi-threading, and user input settings.

Given my material science and computational modeling background, I wanted to test this application by constructing knowledge graphs from documentation of ABAQUS, an FEA engineering software, and some technical data sheets of carbon fibers and polymers. The overall accuracy of using the local LLMs could be better, considering the complexity of this dataset. Future articles will explore learnings from benchmark studies using different models for embedding and inference. Nevertheless, I am eager to build more complex knowledge graphs from scientific journals and data in this field, test advanced engineering code generation tasks, and utilize a conversational assistant to brainstorm scientific topics within my expertise. The application looks like this.

Main application UI with example queries. The last two have the same query, but the first is a global search, while the second is a local one.

Widget settings to switch between local and global search, set community levels, and generation length.

The development was done in a Linux environment using the Windows Subsystem for Linux (WSL) and Visual Studio Code on a Windows 11 PC with an i9 13th Gen processor, 64 GB RAM, and 24 GB Nvidia RTX 4090. For the best experience in developing and testing this app, it is recommended to use a Linux distribution or WSL. I have not tested this on a native Windows environment. For guidelines on installing WSL and setting up Python and Conda environments, please refer to this article (here). Additional references and relevant information are provided at the end of this article.

Here is the link to the source code repository. Now, let's get started!!

Install model dependencies & clone repository.

Install language models from Ollama for inference and embedding

```
# Mistral for GraphRAG Inference
ollama pull mistral

# Nomic-Embed-Text for GraphRAG Embedding
ollama pull nomic-embed-text

# LLama3 for Autogen Inference
ollama pull llama3

# Host Ollama on a local server: http://localhost:11434
ollama serve
```

Create a conda environment and install these dependencies

```
# Create and activate a conda environment
conda create -n RAG_agents python=3.12
conda activate RAG_agents

# Lite-LLM proxy server for Ollama
pip install 'litellm[proxy]'

# Install Ollama
pip install ollama

# Microsoft AutoGen
pip install pyautogen "pyautogen[retrievechat]"

# Microsoft GraphRAG
pip install graphrag

# Text-Token Encoder-Decoder
pip install tiktoken

# Chainlit Python application
pip install chainlit

# Clone my Git-hub repository
git clone https://github.com/karthik-codex/autogen_graphRAG.git

# (BONUS) To Convert PDF files to Markdown for GraphRAG
pip install marker-pdf

# (BONUS) Only if you installed Marker-pdf since it removes GPU CUDA support by
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c nvi
```

You will find the following files in my GitHub repository.

1. `/requirements.txt` — *Contains a list of all the above packages*
2. `/utils/settings.yaml` — *Contains the LLM config for using Mistral 7B and Nomic-Text-Embedding from Ollama for GraphRAG offline embedding and indexing. You will use this file to replace the one created when you initialize GraphRAG in your working directory for the first time.*
3. `/utils/chainlit_agents.py` — *Contains class definitions that include AutoGen's assistant and user proxy agents. This allows multiple agents to be tracked and their messages displayed in the UI. (Shout out to the Chainlit team for building the template).*
4. `/utils/embedding.py` —*Contains the modified embedding functions for GraphRAG embedding for local search queries using Ollama. You will use this file to replace the one inside GraphRAG package (more info below)*
5. `utils/openai_embeddings_llm.py` —*Contains the modified embedding functions for GraphRAG indexing and embedding using Ollama. You will use this file to replace the one inside GraphRAG package (more info below).*
6. `/appUI.py` —*Contains the main asynchronous functions to set up agents, define GraphRAG search functions, track and handle messages, and display them inside Chainlit UI.*
7. `/utils/pdf_to_markdown.py` — *Bonus file containing functions to convert PDF files to markdown files for GraphRAG ingestion.*

Create a GraphRAG knowledge base.

Initialize GraphRAG in the root folder of the repository

```
#make a new folder "input" to place your input files for GraphRAG (.txt or .md)
mkdir -p ./input

# Initialize GraphRAG to create the required files and folders in the root dir
python -m graphrag.index --init  --root .

# Move the settings.yaml file to replace the one created by GraphRAG --init
mv ./utils/settings.yaml ./
```

Configure GraphRAG settings to support local models from Ollama

Below is a snippet from `settings.yaml` illustrating the configuration of LLMs for creating indexes and embeddings. GraphRAG requires a 32k context length for indexing, making Mistral the chosen model. For embeddings, Nomic-embed-text is selected, although you can experiment with other embeddings from Ollama. No need to set `${GRAPHRAG_API_KEY}`, as access is not required to these local models' endpoints.

```
encoding_model: cl100k_base
skip_workflows: []
llm:
  api_key: ${GRAPHRAG_API_KEY}
  type: openai_chat # or azure_openai_chat
  model: mistral
  model_supports_json: true
  api_base: http://localhost:11434/v1
.
.
.
embeddings:
  async_mode: threaded # or asyncio
  llm:
    api_key: ${GRAPHRAG_API_KEY}
    type: openai_embedding # or azure_openai_embedding
    model: nomic_embed_text
    api_base: http://localhost:11434/api
.
.
.
input:  #Change input file pattern to.md, or .txt
  type: file # or blob
  file_type: text # or csv
  base_dir: "input"
```

```
    file_encoding: utf-8
    file_pattern: ".*\\.md$"
```

You can specify the folder containing the input files in the “input” folder in the root directory. Both text and markdown files can be used. You can use the `/utils/pdf_to_markdown.py` to convert your PDFs to markdown files that are then placed inside the “input” folder. Handling multiple file formats has not been figured out, but it is a solvable issue.

Before running GraphRAG to index, create embeddings, and perform local queries, you must modify the Python files `openai_embeddings_llm.py` and `embedding.py` located within the GraphRAG package. Without this modification, GraphRAG will throw an error when creating embeddings, as it won't recognize "nomic-embed-text" as a valid embedding model from Ollama. In my setup, these files are located at `/home/karthik/miniconda3/envs/RAG_agents/lib/python3.12/site-packages/graphrag/llm/openai/openai_embeddings_llm.py` and `/home/karthik/miniconda3/envs/RAG_agents/lib/python3.12/site-packages/graphrag/query/llm/oai/embedding.py`

You can locate these files using the command `sudo find / -name openai_embeddings_llm.py`.

Create embeddings and knowledge graphs.

Lastly, we create the embeddings and test the knowledge graph using the global or local search method. After completing the embedding process, you can find the output artifacts (.parquet files) and reports (.json and .logs) in the “output” folder of your GraphRAG working directory, which is the root folder in this instance.

```
# Create knowledge graph - this takes some time
python -m graphrag.index --root .
```

```
# Test GraphRAG
python -m graphrag.query --root . --method global "<insert your query>"
```

Start the Lite-LLM server and run the app from the terminal

Below is the command to initialize the server before running the app. I chose Llama3:8b to test this app. You can use larger models if your hardware permits. More information on Lite-LLM can be found at this link. Now, you are ready to run the application from another terminal. Make sure you are in the right conda environment.

```
# start server from terminal
litellm --model ollama_chat/llama3

# run app from another terminal
chainlit run appUI.py
```

Breakdown: Core components of appUI.py

Import python libraries

```
import autogen
from rich import print
import chainlit as cl
from typing_extensions import Annotated
from chainlit.input_widget import (
   Select, Slider, Switch)
from autogen import AssistantAgent, UserProxyAgent
from utils.chainlit_agents import ChainlitUserProxyAgent, ChainlitAssistantAgent
from graphrag.query.cli import run_global_search, run_local_search
```

You will notice two classes being imported from *chainlit_agents*. These wrapper classes for AutoGen agents enable Chainlit to track their conversations and handle termination or other user inputs. You can read more about this here.

Configure AutoGen agents

The AutoGen agents utilize models from Ollama via the Lite-LLM proxy server. This is necessary because AutoGen does not support function calling through non-OpenAI inference models. The proxy server enables using Ollama models for function calling and code execution.

```
# LLama3 LLM from Lite-LLM Server for Agents #
llm_config_autogen = {
    "seed": 40,  # change the seed for different trials
    "temperature": 0,
    "config_list": [{"model": "litellm",
                     "base_url": "http://0.0.0.0:4000/",
                     'api_key': 'ollama'},
    ],
    "timeout": 60000,
}
```

Instantiate agents and input user settings at the start of the chat

I created three Chainlit widgets (switch, select, and slider) as user settings to choose the GraphRAG search type, community level, and content generation type. When turned ON, the switch widget uses the GraphRAG local search method for querying. The select options for content generation include “prioritized list,” “single paragraph,” “multiple paragraphs,” and “multiple-page report.” The slider widget selects the community generation level with options 0, 1, and 2. You can read more about the GraphRAG communities here.

```
@cl.on_chat_start
async def on_chat_start():
  try:
    settings = await cl.ChatSettings(
            [
                Switch(id="Search_type", label="(GraphRAG) Local Search", initia
                Select(
                    id="Gen_type",
                    label="(GraphRAG) Content Type",
                    values=["prioritized list", "single paragraph", "multiple pa
                    initial_index=1,
                ),
```

```
                Slider(
                    id="Community",
                    label="(GraphRAG) Community Level",
                    initial=0,
                    min=0,
                    max=2,
                    step=1,
                ),

            ]
        ).send()

    response_type = settings["Gen_type"]
    community = settings["Community"]
    local_search = settings["Search_type"]

    cl.user_session.set("Gen_type", response_type)
    cl.user_session.set("Community", community)
    cl.user_session.set("Search_type", local_search)

    retriever   = AssistantAgent(
       name="Retriever",
       llm_config=llm_config_autogen,
       system_message="""Only execute the function query_graphRAG to look for co
                    Output 'TERMINATE' when an answer has been provided.""",
       max_consecutive_auto_reply=1,
       human_input_mode="NEVER",
       description="Retriever Agent"
    )

    user_proxy = ChainlitUserProxyAgent(
        name="User_Proxy",
        human_input_mode="ALWAYS",
        llm_config=llm_config_autogen,
        is_termination_msg=lambda x: x.get("content", "").rstrip().endswith("TER
        code_execution_config=False,
        system_message='''A human admin. Interact with the retriever to provide
        description="User Proxy Agent"
    )

    print("Set agents.")

    cl.user_session.set("Query Agent", user_proxy)
    cl.user_session.set("Retriever", retriever)

    msg = cl.Message(content=f"""Hello! What task would you like to get done tod
                     """,
                     author="User_Proxy")
    await msg.send()

    print("Message sent.")

  except Exception as e:
    print("Error: ", e)
    pass
```

I chose not to use the Chainlit wrapper class for the retriever assistant agent. This allowed me to disable tracking of the retriever's output and directly capture the response from the GraphRAG function. The reason is that when the response passes through the retriever, the text loses its formatting, including spaces and paragraph indents. This issue was especially noticeable when generating multi-page reports with main and sub-headings. I could preserve the original formatting by bypassing the Chainlit wrapper and directly retrieving the output from the GraphRAG function. You will see how I achieved this below.

Update changes in input settings

This function detects any changes made to the select, switch, and slider widgets from settings so it can reflect those changes in the subsequent queries.

```
@cl.on_settings_update
async def setup_agent(settings):
    response_type = settings["Gen_type"]
    community = settings["Community"]
    local_search = settings["Search_type"]
    cl.user_session.set("Gen_type", response_type)
    cl.user_session.set("Community", community)
    cl.user_session.set("Search_type", local_search)
    print("on_settings_update", settings)
```

Update UI with incoming messages from agents and the user.

This is the core part of the application, which creates a group chat with two agents, defines a function "state_transition" to manage the conversation sequence, and provides the asynchronous RAG query function.

You will notice `INPUT_DIR ,ROOT_DIR, RESPONSE_TYPE, COMMUNTIY` parameters that are passed into the local and global search GraphRAG query functions

based on the bool parameter `LOCAL_SEARCH`. The `ROOT_DIR,` is set to `'.'` — pay attention to this if you initialized GraphRAG in a different directory.

The asynchronous function “query_graphRAG” calls the GraphRAG global or local search method. You will notice the line `await cl.Message(content=result.response).send()` inside the `async def query_graphRAG` function that directly retrieves the output from the RAG query and preserves the text formatting of the retrieved content.

```
@cl.on_message
async def run_conversation(message: cl.Message):
    print("Running conversation")
    CONTEXT = message.content

    MAX_ITER = 10
    INPUT_DIR = None
    ROOT_DIR = '.'
    RESPONSE_TYPE = cl.user_session.get("Gen_type")
    COMMUNITY = cl.user_session.get("Community")
    LOCAL_SEARCH = cl.user_session.get("Search_type")

    print("Setting groupchat")

    retriever   = cl.user_session.get("Retriever")
    user_proxy  = cl.user_session.get("Query Agent")

    def state_transition(last_speaker, groupchat):
        messages = groupchat.messages
        if last_speaker is user_proxy:
            return retriever
        if last_speaker is retriever:
            if messages[-1]["content"].lower() not in ['math_expert','physics_ex
                return user_proxy
            else:
                if messages[-1]["content"].lower() == 'math_expert':
                    return user_proxy
                else:
                    return user_proxy
        else:
            pass
            return None

    async def query_graphRAG(
          question: Annotated[str, 'Query string containing information that you
                          ) -> str:
        if LOCAL_SEARCH:
            result = run_local_search(INPUT_DIR, ROOT_DIR, COMMUNITY ,RESPONSE_T
        else:
```

```
            result = run_global_search(INPUT_DIR, ROOT_DIR, COMMUNITY ,RESPONSE_
        await cl.Message(content=result).send()
        return result

    for caller in [retriever]:
        d_retrieve_content = caller.register_for_llm(
            description="retrieve content for code generation and question answe
        )(query_graphRAG)

    for agents in [user_proxy, retriever]:
        agents.register_for_execution()(d_retrieve_content)

    groupchat = autogen.GroupChat(
        agents=[user_proxy, retriever],
        messages=[],
        max_round=MAX_ITER,
        speaker_selection_method=state_transition,
        allow_repeat_speaker=True,
    )
    manager = autogen.GroupChatManager(groupchat=groupchat,
                                       llm_config=llm_config_autogen,
                                       is_termination_msg=lambda x: x.get("conte
                                       code_execution_config=False,
                                       )

# -------------------- Conversation Logic. Edit to change your first message bas
    if len(groupchat.messages) == 0:
      await cl.make_async(user_proxy.initiate_chat)( manager, message=CONTEXT, )
    elif len(groupchat.messages) < MAX_ITER:
      await cl.make_async(user_proxy.send)( manager, message=CONTEXT, )
    elif len(groupchat.messages) == MAX_ITER:
      await cl.make_async(user_proxy.send)( manager, message="exit", )
```

For this application, we only need two agents. You can add/modify agents and configure the “state_transition” function to orchestrate speaker selection in conversations for more complex workflows.

Final Thoughts

This is my first venture into AI agents, LLMs, and RAGs, and I dove straight into creating this implementation over the past few weeks, bypassing many basics. While this implementation is imperfect, it is an excellent template for developing more complex applications. It lays a solid foundation for integrating multiple functions and coding agents and should enable you to

build sophisticated workflows, customize agent interactions, and enhance functionality as needed.

> ***About Me**: I am a lead modeling engineer at Eaton Research Labs, Southfield, MI, USA. I explore, develop tools, and write about things at the intersection of computational mechanics, material science, engineering, language models, and generative AI.*

If you want to stay updated, follow me on my socials below.

Socials: LinkedIn, GitHub, Source Code

Some Useful References

1. https://medium.com/@datadrifters/autogen-litellm-and-open-source-llms-c4c6bc8fa9c5
2. https://medium.com/@rajib76.gcp/memory-default-compute-fallback-5ff4287d47e6
3. https://medium.com/generative-ai/graphrag-the-rag-approach-by-microsoft-e1abc7eb9fba
4. https://docs.chainlit.io/get-started/overview
5. https://medium.com/@antoineross/autogen-web-application-using-chainlit-8c5ebf5a4e75

Autogen Ollama Chainlit Rags Graphrag

Written by Karthik Rajan, Ph.D

970 Followers · 10 Following

Follow

I explore, develop tools & write about things at the intersection of computational mechanics, material science, engineering, & generative AI.

Responses (17)

See all responses