Multi-agent RAG with AutoGen: Build locally with Granite

Can you build agentic workflows without needing extremely large, costly large language models (LLMs)? The answer is yes. In this tutorial, we will demonstrate how to build a multi-agent RAG system locally with AutoGen by using IBM® Granite™.

Agentic RAG overview

Retrieval-augmented generation (RAG) is an effective way of providing an LLM with additional datasets from various data sources without the need for expensive fine-tuning. Similarly, agentic RAG leverages an AI agent's ability to plan and execute subtasks along with the retrieval of relevant information to supplement an LLM's knowledge base. This ability allows for the optimization and greater scalability of RAG applications compared to traditional chatbots. No longer do we need to write complex SQL queries to extract relevant data from a knowledge base.

The future of agentic RAG is multi-agent RAG, where several specialized agents collaborate to achieve optimal latency and efficiency. We will demonstrate this collaboration by using a small, efficient model such as Granite 3.2 and combining it with a modular agent architecture. We will use multiple specialized "mini agents" that collaborate to achieve tasks through adaptive planning and tool or function calling. Like humans, a team of agents, or a multi-agent system, often outperforms the heroic efforts of an individual, especially when they have clearly defined roles and effective communication.

For the orchestration of this collaboration, we can use AutoGen (AG2) as the core framework to manage workflows and decision-making, alongside other tools like Ollama for local LLM serving and Open WebUI for interaction. AutoGen is a framework for creating multi-agent AI applications developed by Microsoft.[1] Notably, every component leveraged in this tutorial is open source. Together, these tools enable you to build an AI system that is both powerful and privacy-conscious, without leaving your laptop.

Multi-agent architecture: When collaboration beats competition

Our Granite retrieval agent relies on a modular architecture in which each agent has a specialized role. Like humans, agents perform best when they have targeted instructions and just enough context to make an informed decision. Too much extraneous information, such as an unfiltered chat history, can create a "needle in the haystack" problem, where it becomes increasingly difficult to decipher signal from noise.

In this agentic AI architecture, the agents work together sequentially to achieve the goal. Here is how the generative AI system is organized:

Planner agent: Creates the initial high-level plan, once in the beginning of the workflow. For example, if a user asks, "What are comparable open source projects to the ones my team is using?" then, the agent will put together a step-by-step plan that might look something like this: "1. Search team documents for open source technologies. 2. Search the web for similar open source projects to the ones found in step 1." If any of these steps fail or provide insufficient results, the steps can be later adapted by the reflection agent.

Research Aasistant: The research assistant is the workhorse of the system. It takes in and executes instructions such as “Search team documents for open source technologies.” For step 1 of the plan, it uses the initial instruction from the planner agent. For subsequent steps, it also receives curated context from the outcomes of previous steps.

For example, if asked to “Search the web for similar open source projects,” it will also receive the output from the previous document search step. Depending on the instruction, the research assistant can use tools like web search or document search, or both, to fulfill its task.

Step critic: The step critic is responsible for deciding whether the output of the previous step satisfactorily fulfilled the instruction it was given. It receives two pieces of information: the single-step instruction that was just executed and the output of that instruction. Having a step critic weigh in on the conversation brings clarity around whether the goal was achieved, which is needed for the planning of the next step.

Goal judge: The goal judge determines whether the ultimate objective has been met, based on all of the requirements of the provided goal, the plans drafted to achieve it, and the information gathered so far. The output of the judge is either "YES" or "NOT YET" followed by a brief explanation that is no longer than one or two sentences.

Reflection agent: The reflection agent is our executive decision-maker. It decides what step to take next, whether that is encroaching onto the next planned step, pivoting course to make up for mishaps or confirming that the goal has been completed. Like a real-life CEO, it performs its best decision-making when it has a clear goal in mind and is presented with concise findings on the progress that has or has not been made to reach that goal. The output of the reflection agent is either the next step to take or the instructions to terminate if the goal has been reached. We present the reflection agent with the following items:

- The goal
- The original plan
- The last step that was executed
- The result of the last step indicating success or failure
- A concise sequence of previously executed instructions (just the instructions, not their output)

Presenting these items in a structured format makes it clear to our decision maker what has been done so that it can decide what needs to happen next.

Report Generator: Once the goal is achieved, the Report Generator synthesizes all findings into a cohesive output that directly answers the original query. While each step in the process generates targeted outputs, the Report Generator ties everything together into a final report.

Leveraging open source tools

For beginners, it can be difficult to build an agentic AI application from scratch. Hence, we will use a set of open source tools. The Granite Retrieval Agent integrates multiple tools for agentic RAG.

Open WebUI: The user interacts with the system through an intuitive chat interface hosted in Open WebUI. This interface acts as the primary point for submitting queries (such as “Fetch me the latest news articles pertaining to my project notes”) and viewing the outputs.

Python-based agent (AG2 framework): At the core of the system is a Python-based agent built by using AutoGen (AG2). This agent coordinates the workflow by breaking down tasks and dynamically

calling tools to execute steps.

The agent has access to two primary tools:

- Document search tool: Fetches relevant information from a vector database containing uploaded project notes or documents stored as embeddings. This vector search leverages the built-in documental retrieval APIs inside Open WebUI, rather than setting up an entirely separate data store.

- Web search tool: Performs web-based searches to gather external knowledge and real-time information. In this case, we are using SearXNG as our metasearch engine.

Ollama: The IBM Granite 3.2 LLM serves as the language model powering the system. It is hosted locally by using Ollama, ensuring fast inference, cost efficiency and data privacy. If you are interested in running this project with larger models, API access through IBM watsonx.ai® or OpenAI, for example, is preferred. This approach, however, requires a watsonx.ai or OpenAI API key. Instead, we use locally hosted Ollama in this tutorial.

Other common open source, agent frameworks not covered in this tutorial include LangChain, LangGraph and crewAI.

Steps

Detailed setup instructions as well as the entire project can be viewed on the IBM Granite Community GitHub. The Jupyter Notebook version of this tutorial can be found on GitHub as well.

The following steps provide a quick setup for the Granite retrieval agent.

Step 1: Install Ollama

Step 2. Build a simple agent (optional)

A low-code solution for building agentic workflows is AutoGen Studio. However, building a simple agent ourselves can help us better understand the setup of this complete multi-agent RAG project. To continue, set up a Jupyter Notebook in your preferred integrated development environment (IDE) and activate a virtual environment by running the following commands in your terminal.
python3.11 -m venv venv source venv/bin/activate
Powered by Granite

We'll need a few libraries and modules for this simple agent. Make sure to install and import the following ones.
```
!pip install -qU langchain chromadb tf-keras pyautogen "ag2[ollama]" sentence_transformers
```
Powered by Granite
import getpass from autogen.agentchat.contrib.retrieve_assistant_agent import AssistantAgent from autogen.agentchat.contrib.retrieve_user_proxy_agent import RetrieveUserProxyAgent
Powered by Granite

There are several configuration parameters to set locally to invoke the correct LLM that we pulled by using Ollama.
ollama_llm_config = { "config_list": [{ "model": "granite3.2:8b", "api_type": "ollama", }], }

Powered by Granite

We can pass these configuration parameters in the `llm_config` parameter of the `AssistantAgent` class to instantiate our first AI agent.
assistant = AssistantAgent(name="assistant", system_message="You are a helpful assistant.", llm_config=ollama_llm_config,)
Powered by Granite

This agent uses Granite 3.2 to synthesize the information returned by the `ragproxyagent` agent. The document we provide to the RAG agent as additional context is the raw README Markdown file found in the AutoGen repository on GitHub. Additionally, we can pass a new dictionary of configurations specific to the retrieval agent. Some additional keys that you might find useful are `vector_db`, `chunk_token_size` and `embedding_model`.

For a full list of configuration keys, refer to the official documentation.
ragproxyagent = RetrieveUserProxyAgent(name="ragproxyagent", max_consecutive_auto_reply=3, is_termination_msg=lambda msg: msg.get("content") is not None or "TERMINATE" in msg["content"], system_message = "Context retrieval assistant.", retrieve_config={ "task": "qa", "docs_path": "https://raw.githubusercontent.com/microsoft/autogen/main/README.md", "get_or_create": True, "collection_name": "autogen_docs", "overwrite": True }, code_execution_config=False, human_input_mode="NEVER",)
Powered by Granite

Now, we can initiate a chat with our RAG agent to ask a question that pertains to the document provided as context.
qs = "What languages does AutoGen support?" result = ragproxyagent.initiate_chat(assistant, message=ragproxyagent.message_generator, problem=qs) print(result)
Powered by Granite

Please note, the README file Markdown content has been redacted from the output for brevity.

Output:

Trying to create collection.

2025-07-21 12:20:36,125 - autogen.agentchat.contrib.retrieve_user_proxy_agent - INFO - Found 1 chunks.
2025-07-21 12:20:36,129 - autogen.agentchat.contrib.vectordb.chromadb - INFO - No content embedding is provided. Will use the VectorDB's embedding function to generate the content embedding.

VectorDB returns doc_ids: [['8e9131c7']]
Adding content of doc 8e9131c7 to context.

***ragproxyagent** (to assistant):*

You're a retrieve augmented chatbot. You answer user's questions based on your own knowledge and the context provided by the user.
If you can't answer the question with or without the current context, you should reply exactly `UPDATE CONTEXT`.
You must give as short an answer as possible.

User's question is: What languages does AutoGen support?

Context is:

[REDACTED]

---

assistant (to ragproxyagent):

The provided text appears to be a README file for the AutoGen project, an open-source initiative by Microsoft. Here's a summary of its content:

1. Project Overview: AutoGen is a system for creating and managing complex, distributed applications. It supports multiple languages (C#, Python) and provides a runtime environment for deploying and scaling these applications.
2. Key Components: - AutoGen Core: The core library containing the fundamental classes and interfaces for building AutoGen applications. - AutoGen Agent: A runtime component responsible for managing the lifecycle of AutoGen applications. - AutoGen Runtime Gateway: A service that facilitates communication between agents and enables load balancing, scaling, and fault tolerance.
3. Languages Supported: C# and Python are currently supported.
4. Getting Started: The README provides instructions on how to install the necessary packages, create a new project, and build/run an AutoGen application.
5. Documentation: Links to detailed documentation for reference, including API references, guides, and tutorials.
6. Community & Contribution: Guidelines for contributing to the project, including information on issue tracking, pull requests, and coding standards.
7. Legal Notices: Licensing information and trademark notices.
8. Support & FAQ: Information on how to ask questions, report issues, and find answers to common queries.

The README also includes a table summarizing the available packages for each supported language (C# and Python) and their respective package managers (NuGet and PyPI). This makes it easy for developers to quickly identify the necessary components for getting started with AutoGen in their preferred language.

TERMINATING RUN (601a53dc-8a5d-4e19-8503-1517fe3c7634): Termination message condition on agent 'ragproxyagent' met

Great! Our assistant agent and RAG agent successfully synthesized the additional context to correctly respond to the user query with the programming languages currently supported by AutoGen. You can think of this as a group chat between agents exchanging information. This example is a simple demonstration of implementing agentic RAG locally with AutoGen.

Step 3. Install Open WebUI

Now, let’s move on to building a more advanced agentic RAG system. In your terminal, install and run Open WebUI.
pip install open-webui open-webui serve
Powered by Granite

Step 4. Set up web search

For web search, we will leverage the built-in web search capabilities in Open WebUI.

Open WebUI supports a number of search providers. Broadly, you can either use a 3rd-party application programming interface (API) service, for which you will need to obtain an API key, or you can locally set up a SearXNG Docker container. In either case, you will need to configure your search provider in the Open WebUI console.

This configuration, either a pointer to SearXNG or input of your API key, is under **Admin Panel > Settings > Web Search** in the Open WebUI console.

Please refer to the instructions in the Open WebUI documentation for more detailed instructions.

Step 5. Import the agent into Open WebUI

1. In your browser, go to http://localhost:8080/ to access Open Web UI. If it is your first time opening the Open WebUI interface, register a username and password. This information is kept entirely local to your machine.

2. After logging in, click the icon on the lower left side where your username is. From the menu, click Admin panel.

3. In the **Functions** tab, click + to add a new function.

4. Give the function a name, such as "Granite RAG Agent," and a description, both of str type.

5. Paste the granite_autogen_rag.py Python script into the text box provided, replacing any existing content.

6. Click **Save** at the bottom of the screen.

7. Back on the **Functions** page, make sure that the agent is toggled to **Enabled**.

8. Click the gear icon next to the enablement toggle to customize any settings such as the inference endpoint, the SearXNG endpoint or the model ID.

Now, your brand-new AutoGen agent shows up as a model in the Open WebUI interface. You can select it and provide it with user queries.

Step 6. Load documents into Open WebUI

1. In Open WebUI, navigate to **Workspace > Knowledge.**
2. Click + to create a new collection.
3. Upload documents for the Granite retrieval agent to query.

Step 7. Configure Web Search in Open WebUI

To set up a search provider (for example, SearXNG), follow this guide.

The configuration parameters are as follows:

Parameter	Description	Default Value

task_model_id	Primary model for task execution	granite3.2:8b
vision_model_id	Vision model for image analysis	granite-vision3.2:2b
openai_api_url	API endpoint for OpenAI-style model calls	http://localhost:11434
openai_api_key	API key for authentication	ollama
vision_api_url	Endpoint for vision-related tasks	http://localhost:11434
model_temperature	Controls response randomness	0
max_plan_steps	Maximum steps in agent planning	6

Note: These parameters can be configured through the gear icon in the "Functions" section of the Open WebUI Admin Panel after adding the function.

Step 8. Query the agentic system

The Granite retrieval agent performs AG2-based RAG by querying local documents and web sources, performing multi-agent task planning and enforcing adaptive execution. Start a chat and provide your agentic system with a query related to the documents provided to see the RAG chain in action.

Summary

A multi-agent setup enables the creation of practical, usable tools by getting the most out of moderately sized, open source models like Granite 3.2. This agentic RAG architecture, built with fully open source tools, can serve as a launching point to design and customize your question answering agents and AI algorithms. It can also be used outside of the box for a wide array of use cases. In this tutorial, you had the opportunity to delve into simple and complex agentic systems, leveraging the capabilities of AutoGen. The Granite LLM was invoked by using Ollama, allowing for a fully local exploration of these systems. As a next step, consider integrating more custom tools into your agentic system.