Deploy an IT support LangGraph ReAct agent with IBM Granite on watsonx.ai

In this tutorial, you will build a ReAct (Reasoning and Action) AI agent with the open-source LangGraph framework by using an IBM Granite model through the IBM® watsonx.ai® API in Python. The use case is to manage existing IT support tickets and to create new ones.

What is a ReAct agent?

An artificial intelligence (AI) agent refers to a system or program that is capable of autonomously performing tasks on behalf of a user or another system by designing its agent workflow and using available tools. Generative AI agents use the advanced natural language processing (NLP) techniques of large language models (LLMs) to comprehend and respond to user inputs step-by-step and determine when to call on external tools. A core component of AI agents is reasoning. Upon acquiring new information through tool calling, human intervention or other agents, the reasoning paradigm guides the agent’s next steps.

With each action and each tool response, the ReAct (Reasoning and Action) paradigm instructs agents to "think" and plan their next steps. This step-by-step, slow reasoning, gives us insight into how the agent uses updated context to formulate conclusions. Because this process of reflection is continuous, it is often referred to as a think-act-observe loop and is a form of chain-of-thought prompting.

Using LangGraph to build ReAct agents

This tutorial will use the LangGraph framework, an open source AI agent framework designed to build, deploy and manage complex generative AI agent workflows. The prebuilt create_react_agent function provided by LangGraph is an easy way to build a simple, custom agent. Simple ReAct agents, like the one in this tutorial as depicted in Figure 1, are composed of two nodes. One node is responsible for calling the model and the other node is for using tools. Common tools include the prebuilt LangChain Wikipedia tool, DuckDuckGoSearchRun tool and even retrieval-augmented generation (RAG). In cases with complex action input, another node can be added, as seen in Figure 2. This additional node would serve the purpose of helping ensure the agent returns structured output.

Within LangGraph, the “state” feature serves as a memory bank that records and tracks all the valuable information processed by each iteration of the AI system. These stateful graphs allow agent's to recall past information and valuable context. The cyclic structure of the ReAct graph is leveraged when the outcome of one step depends on previous steps in the loop. The nodes, or "actors," in the graph encode agent logic and are connected by edges. Edges are essentially Python functions that determine the next node to execute depending on the current state.

Prerequisites

You need an IBM Cloud® account to create a watsonx.ai™ project.

Steps

In order to use the watsonx application programming interface (API), you will need to complete the following steps. Note, you can also access this tutorial on GitHub.

Step 1. Generate your watsonx.ai credentials

1. Log in to watsonx.ai using your IBM Cloud account.

2. Create a watsonx.ai Runtime service instance (select your appropriate region and choose the Lite plan, which is a free instance).

3. Generate an application programming interface (API) key.

Step 2. Set up the project in your IDE

To easily get started with deploying agents on watsonx.ai, clone this GitHub repository and access the IT support ReAct agent project. You can run the following command in your terminal to do so.

```
git clone git@github.com:IBM/ibmdotcom-tutorials.git cd react-agent-langgraph-it-support/base/langgraph-react-agent/
```

Next, install poetry if you do not already have it installed. Poetry is a tool for managing Python dependencies and packaging.

```
pipx install --python 3.11 poetry
```

Then, activate your virtual environment.

```
source $(poetry -q env use 3.11 && poetry env info --path)/bin/activate
```

Rather than using the pip install command, the poetry package allows us to add dependencies by running the following command in the terminal. This step installs the repository dependencies reflected in the pyproject.toml file to your separate virtual environment.

```
poetry install
```

Adding a working directory to PYTHONPATH is necessary for the next steps. In your terminal execute:

```
export PYTHONPATH=$(pwd):${PYTHONPATH}
```

To set up your environment, follow along with the instructions in the README.md file on Github. This set up requires several commands to be run on your IDE or command line.

Step 3. Set the environment variables

In theconfig.toml file, you will find the following blank credentials that must be filled in before attempting to deploy your agent. Yourwatsonx_apikey andwatsonx_url were initialized in step 1 of this tutorial. Next, follow along with the simple form found on the Developer Access page to select your deployment space or create a new one. There, you can retrieve yourspace_id needed to connect our agent to the watsonx.ai deployment. Lastly, yourmodel_id is set to the IBM Granite 3.2 model.

```
[deployment]   watsonx_apikey = “”   watsonx_url = “”  # should follow the format:
`https://{REGION}.ml.cloud.ibm.com`   space_id = “” # found in the “Manage” tab of your
Deployment or in the Developer Access page here: https://dataplatform.cloud.ibm.com/developer-
```

```
access [deployment.custom] # during creation of deployment additional parameters can be provided
inside `CUSTOM` object for further referencing # please refer to the API docs:
https://cloud.ibm.com/apidocs/machine-learning-cp#deployments-create   model_id = “ibm/granite-3-
2-8b-instruct”   thread_id = “thread-1” # More info here: https://langchain-ai.github.io/langgraph/how-
tos/persistence/
```

Step 4. Upload your data to IBM Cloud Object Storage

Our agent requires a data source to provide up-to-date information and add new data. We will store our data file in IBM Cloud® Object Storage.

1. First, log in to IBM Cloud. Then, create a new project.
2. In the left-side menu, select Resource list. Using the Create resource button, create a new Cloud Object Storage instance or simply use this link.
3. Open your newly created IBM Cloud Storage Instance, create a new bucket. For this tutorial, you can select the Smart Tier which is the free tier. When directed, upload your file. For the sample file, refer to the tickets.csv file in the GitHub repository linked in step 2.

Step 5. Establish your data connection

To provide the ReAct agent with IT ticket management functionality, we must connect to our data source in IBM Cloud Object Storage. For this step, we can use the ibm_boto3 library.

In tools.py , the COS_ENDPOINT , COS_INSTANCE_CRN , BUCKET_NAME and CSV_FILE_NAME must be filled in with the appropriate information using the bucket details found in your Cloud Object Storage instance under the Configuration tab.

```
  COS_ENDPOINT = ""      #find in your COS bucket configuration   COS_INSTANCE_CRN = ""
 #find in your COS bucket configuration   BUCKET_NAME = ""       #find in your COS bucket
configuration   CSV_FILE_NAME = "filename.csv" #you can use the provided tickets.csv sample file
  cos = ibm_boto3.client(      "s3",      ibm_api_key_id=dep_config["watsonx_apikey"],
ibm_service_instance_id=COS_INSTANCE_CRN,      config=Config(signature_version="oauth"),
endpoint_url=COS_ENDPOINT,   )
```

Step 6. Create your custom tools

Our agent will be able to both read and write data in our file. First, let's create the tool to read data using the LangChain@tool decorator.

We have added thisfind_tickets tool to thetools.py file. This tool retrieves the data object from Cloud Object Storage and returns it as a Pandas dataframe. Otherwise, an exception is thrown.

```
  @tool   def find_tickets():      """Returns a list of of all tickets."""      try:          response =
cos.get_object(Bucket=BUCKET_NAME, Key=CSV_FILE_NAME)          csv_data =
pd.read_csv(response['Body'])          print("Ticket file loaded successfully:")          return csv_data
except Exception as e:          print(f"Error loading file from COS: {e}")          return None
```

Next, we have added the create_ticket tool.

```
  @tool  def create_ticket(issue: str, urgency:str):     """Creates a tickets for a customer issue. Request a
detailed explanation of the customer issue and urgency level before creating a ticket.          Args:
issue (str): A description of the issue.        urgency (str): A category value for the level of urgency. Can
```

```
be "low", "medium", or "high".     Returns:     The new ticket.   """   try:     # retrieve the
existing item to reload the contents     response = cos.get_object(Bucket=BUCKET_NAME,
Key=CSV_FILE_NAME)     existing_body_df = pd.read_csv(response['Body'])     new_ticket =
{"issue": issue, "date_added":datetime.now().strftime("%m-%d-%Y"), "urgency":urgency,
"status":"open"}     # Add a new row (i.e. ticket) using loc[]
existing_body_df.loc[len(existing_body_df)] = new_ticket
cos.put_object(Bucket=BUCKET_NAME, Key=CSV_FILE_NAME,
Body=existing_body_df.to_json())     return "New ticket successfully created!"   except Exception
as e:     print("Unable to create new ticket. Please try again.")
```

This tool takes in the description of the issue from the user and the urgency of the issue as its arguments. A new row is added to our file in COS with this information and thus, a new ticket is created. Otherwise, an exception is thrown.

One last tool we must add to our tools.py file is the get_todays_date tool which uses the datetime module to return today's date in MM-DD-YYYY format. This tool will be useful for accessing the current date, which the agent has no other way of retrieving because the LLM was not trained on this data.

```
  @tool  def get_todays_date():   """Returns today's date in MM-DD-YYYY format."""   date =
datetime.now().strftime("%m-%d-%Y")   return date
```

To grant our agent access to these tools, we have added them to the TOOLS list in the extensions module's init.py file. This list should be the contents of your init.py file in the src/langgraph_react_agent directory.

```
  from .tools import (   find_tickets,   get_todays_date,   create_ticket  )  TOOLS = [
find_tickets,    get_todays_date,    create_ticket  ]
```

These tools are imported in the agent.py file and passed to the prebuilt LangGraph create_react_agent function serving as the agent executor. Other parameters include the large language model initialized by using the ChatWatsonx class which allows for tool calling on watsonx.ai, the memory saver and system prompt. Note, some prompts will behave better than others and so, some level of prompt engineering might be required depending on the LLM you choose.

Before deploying your agent, remember to complete all the necessary information in the config.toml file.

Step 7. Chat with your agent

There are three ways to chat with your agent.

Option 1. Query the agent locally

Run the script for local AI service execution.

```
  poetry run python examples/execute_ai_service_locally.py
```

Option 2. Deploy the agent to the built-in chat interface in watsonx.ai

The final option is to access the agent in the Deployments space on watsonx.ai. To do this, select "Deployments" on the left-side menu. Then, select your deployment space, select the "Assets" tab, select youronline ai_service asset, select the asset with the "wx-agent" tag once more and finally, open the "Preview" tab. Now, you can chat with the agent in the interactive chat interface. Each of these 3 options should result in a similar output.

Option 3. Deploy and chat with the agent in your IDE

To run the deployment script, initialize the deployment_id variable in the query_existing_deployment.py file.

The deployment_id of your deployment can be obtained by running the scripts/deploy.py file.

Next, run the deployment script.

```
  poetry run python scripts/deploy.py
```

Then, run the script for querying the deployment.

```
  poetry run python examples/query_existing_deployment.py
```

For the purposes of this tutorial, let's choose option 2 and query our deployed agent on watsonx.ai in the form of an agentic chatbot. Let's provide the agent with some prompts that would require the usage of tools. Upon following the steps listed in Option 3, you should see a chat interface on watsonx.ai. There, we can type our prompt.

First, let's test whether thecreate_ticket tool will be successfully invoked. Let's prompt the agent to create a new ticket.

As you can see in the agent's final answer, the AI system successfully used problem-solving to create a new ticket with the create_ticket tool. Having the visibility into tool calls is helpful for debugging purposes. Now, let's check whether the ticket was successfully added to our data file serving as the agent's knowledge base.

Great! The agent successfully added the ticket to the file.

Conclusion

In this tutorial, you created an agent with the ReAct framework that uses decision making to solve complex tasks such as retrieving and creating support tickets. There are several AI models out there that allow for agentic tool calling such as Google’s Gemini, IBM’s Granite and OpenAI’s GPT-4. In our project, we used an IBM Granite AI model through the watsonx.ai API. The model behaved as expected both locally and when deployed on watsonx.ai. As a next step, check out the LlamaIndex and crewAI multiagent templates available in the watsonx-developer-hub GitHub repository for building AI agents.