Human-in-the-loop (HITL)

Humans are able to asynchronously review and update graph states in LangGraph due to the persistent execution state. By using the state checkpoints after each step, state context can be persisted and the workflow can be paused until human feedback is received.

In this tutorial, we will experiment with the two HITL approaches in LangGraph.

1. **Static interrupts**: Editing the graph state directly at predetermined points *before or after* a specific node is executed. This approach requires the interrupt_before or interrupt_after parameters to be set to a list of node names when compiling the state graph.

2. **Dynamic interrupts**: Interrupting a graph and awaiting user input from *within* a node based on the graph's current state. This approach requires the use of LangGraph's interrupt function.

Step 1. Set up your environment.

While you can choose from several tools, this tutorial walks you through how to set up an IBM account to use a Jupyter Notebook.

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

2. Create a watsonx.ai project.

   You can get your project ID from within your project. Click the **Manage** tab. Then, copy the project ID from the **Details** section of the **General** page. You need this ID for this tutorial.

3. Create a Jupyter Notebook.

   This step opens a Jupyter Notebook environment where you can copy the code from this tutorial. Alternatively, you can download this notebook to your local system and upload it to your watsonx.ai project as an asset. This tutorial is also available on GitHub.

Step 2. Set up a watsonx.ai Runtime instance and API key.

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

2. Generate an API Key.

3. Associate the watsonx.ai Runtime service instance to the project that you created in watsonx.ai.

Step 3. Install and import relevant libraries and set up your credentials.

We need a few libraries and modules for this tutorial. Make sure to import the following ones and if they're not installed, a quick pip installation resolves the problem.
%pip install --quiet -U langgraph langchain-ibm langgraph_sdk langgraph-prebuilt google-search-results

Restart the kernel and import the following packages.
import getpass import uuid from ibm_watsonx_ai import APIClient, Credentials from ibm_watsonx_ai.foundation_models.moderations import Guardian from IPython.display import Image, display from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, AIMessage from langchain_ibm import ChatWatsonx from langgraph.checkpoint.memory import MemorySaver from langgraph.graph import START, END, StateGraph from langgraph.graph.message import add_messages from langgraph.prebuilt import tools_condition, ToolNode from langgraph.types import interrupt, Command from serpapi.google_search import GoogleSearch from typing_extensions import TypedDict from typing import Annotated

To set our credentials, we need the WATSONX_APIKEY and WATSONX_PROJECT_ID that you generated in Step 1. We will also set the WATSONX_URL to serve as the API endpoint.

To access the Google Patents API, we also need a SERPAPI_API_KEY . You can generate a free key by logging into your SerpApi account or registering for one.
WATSONX_APIKEY = getpass.getpass("Please enter your watsonx.ai Runtime API key (hit enter): ")
WATSONX_PROJECT_ID = getpass.getpass("Please enter your project ID (hit enter): ")
WATSONX_URL = getpass.getpass("Please enter your watsonx.ai API endpoint (hit enter): ")
SERPAPI_API_KEY = getpass.getpass("Please enter your SerpAPI API key (hit enter): ")

Before we can initialize our LLM, we can use the Credentials class to encapsulate our passed API credentials.
credentials = Credentials(url=WATSONX_URL, api_key=WATSONX_APIKEY)

Step 4. Instantiate the chat model

To be able to interact with all resources available in watsonx.ai Runtime, you need to set up an APIClient . Here, we pass in our credentials and WATSONX_PROJECT_ID .
client = APIClient(credentials=credentials, project_id=WATSONX_PROJECT_ID)

For this tutorial, we will be using the ChatWatsonx wrapper to set up our chat model. This wrapper simplifies the integration of tool calling and chaining. We encourage you to use the API references in theChatWatsonx official docs for further information. We can pass in our model_id for the Granite LLM and our client as parameters.

Note, if you use a different API provider, you will need to change the wrapper accordingly.
model_id = "ibm/granite-3-3-8b-instruct" llm = ChatWatsonx(model_id=model_id, watsonx_client=client)

Step 5. Define the patent scraper tool

AI agents use tools to fill information gaps and return relevant information. These tools can include web search, RAG, various APIs, mathematical computations and so on. With the use of the Google Patents Api through SerpAPI, we can define a tool for scraping patents. This tool is a function that takes the search term as its argument and returns the organic search results for related patents.
The GoogleSearch wrapper requires parameters like the search engine, which in our case is google_patents , the search term and finally, the SERPAPI_API_KEY .

def scrape_patents(search_term: str):   “””Search for patents about the topic.   Args: search_term: topic to search for   “””   params = {   “engine”: “google_patents”,   “q”: search_term,   “api_key”: SERPAPI_API_KEY   }   search = GoogleSearch(params)   results = search.get_dict()   return results[‘organic_results’]

Next, let’s bind the LLM to the scrape_patents tool by using the bind_tools method.
tools = [scrape_patents] llm_with_tools = llm.bind_tools(tools)

Step 6. First HITL approach: Static interrupts

LangGraph agent graphs are composed of nodes and edges. Nodes are functions that relay, update, and return information. How do we keep track of this information between nodes? Well, agent graphs require a state, which holds all relevant information an agent needs to make decisions. Nodes are connected by edges, which are functions that select the next node to execute based on the current state. Edges can either be conditional or fixed.

Let’s start with creating an AgentState class to store the context of the messages from the user, tools and the agent itself. Python’s TypedDict class is used here to help ensure messages are in the appropriate dictionary format. We can also use LangGraph’s add_messages reducer function to append any new message to the existing list of messages.
class AgentState(TypedDict):   messages: Annotated[list[AnyMessage], add_messages]

Next, define the call_llm function that makes up the assistant node. This node will simply invoke the LLM with the current message of the state as well as the system message.
sys_msg = SystemMessage(content=”You are a helpful assistant tasked with prior art search.”) def call_llm(state: AgentState):   return {“messages”: [llm_with_tools.invoke([sys_msg] + state[“messages”])]

Next, we can define the guardian_moderation function that makes up the guardian node. This node is designed to moderate messages by using a guardian system, to detect and block unwanted or sensitive content. First, the last message is retrieved. Next, a dictionary named detectors is defined that contains the detector configurations and their threshold values. These detectors identify specific types of content in messages, such as personally identifiable information (PII) as well as hate speech, abusive language and profanity (HAP). Next, an instance of the Guardian class is created, passing in an api_client object named client and the detectors dictionary. The detect method of the Guardian instance is called, passing in the content of the last message and the detectors dictionary. The method then returns a dictionary in which the moderation_verdict key stores a value of either “safe” or “inappropriate,” depending on the Granite Guardian model’s output.
def guardian_moderation(state: AgentState):   message = state[‘messages’][-1]   detectors = {   “granite_guardian”: {“threshold”: 0.4},   “hap”: {“threshold”: 0.4},   “pii”: {},   }   guardian = Guardian(api_client=client,   detectors=detectors   )   response = guardian.detect(text=message.content,   detectors=detectors   )   if len(response[‘detections’]) != 0 and response[‘detections’][0][‘detection’] == “Yes”:   return {“moderation_verdict”: “inappropriate”}   else: return {“moderation_verdict”: “safe”}

Now, let’s define the block_message function to serve as a notification mechanism, informing the user that their input query contains inappropriate content and has been blocked.
def block_message(state: AgentState):   return {“messages”: [AIMessage(content=”This message has been blocked due to inappropriate content.”)]

We can now put all of these functions together by adding the corresponding nodes and connecting them with edges that define the flow of the graph.

The graph starts at the guardian node, which calls the guardian_moderation method to detect harmful content before it reaches the LLM and the API. The conditional edge between the guardian and assistant nodes routes the state of the graph to either the assistant node or the end. This position is determined by the output of the guardian_moderation function. Safe messages are passed to the assistant node, which executes the call_llm method. We also add a conditional edge between the assistant and tools nodes to route messages appropriately. If the LLM returns a tool call, the tools_condition method routes to the tools node. Otherwise, the graph routes to the end. This step is part of the ReAct agent architecture because we want the agent to receive the tool output and then react to the change in state to determine its next action.

builder = StateGraph(AgentState) builder.add_node("guardian", guardian_moderation) builder.add_node("block_message", block_message) builder.add_node("assistant", call_llm) builder.add_node("tools", ToolNode(tools)) builder.add_edge(START, "guardian") builder.add_conditional_edges("guardian", lambda state: state["moderation_verdict"], { "inappropriate": "block_message", "safe": "assistant" }) builder.add_edge("block_message", END) builder.add_conditional_edges("assistant", tools_condition,) builder.add_edge("tools", "assistant") memory = MemorySaver()

Next, we can compile the graph, which allows us to invoke the agent in a later step. To persist messages, we can use the MemorySaver checkpointer. To implement the first human oversight approach, static interrupts, we can set the interrupt_before parameter to the assistant node. This means that before the graph routes to the LLM in the assistant node, a graph interruption will take place to allow the human overseeing the agentic workflow to provide feedback.

graph = builder.compile(interrupt_before=["assistant"], checkpointer=memory)

To obtain a visual representation of the agent's graph, we can display the graph flow.

display(Image(graph.get_graph(xray=True).draw_mermaid_png()))