


Human-in-the-loop

 This guide uses the new `interrupt` function.




As of LangGraph 0.2.57, the recommended way to set breakpoints is using the `interrupt` function as it simplifies **human-in-the-loop** patterns.

If you're looking for the previous version of this conceptual guide, which relied on static breakpoints and `NodeInterrupt` exception, it is available [here](#).

A **human-in-the-loop** (or "on-the-loop") workflow integrates human input into automated processes, allowing for decisions, validation, or corrections at key stages. This is especially useful in **LLM-based applications**, where the underlying model may generate occasional inaccuracies. In low-error-tolerance scenarios like compliance, decision-making, or content generation, human involvement ensures reliability by enabling review, correction, or override of model outputs.

Use cases

Key use cases for **human-in-the-loop** workflows in LLM-based applications include:

1.  **Reviewing tool calls:** Humans can review, edit, or approve tool calls requested by the LLM before tool execution.
2.  **Validating LLM outputs:** Humans can review, edit, or approve content generated by the LLM.
3.  **Providing context:** Enable the LLM to explicitly request human input for clarification or additional details or to support multi-turn conversations.

`interrupt`

The `interrupt` function in LangGraph enables human-in-the-loop workflows by pausing the graph at a specific node, presenting information to a human, and resuming the graph with their input. This function is useful for tasks like approvals, edits, or collecting additional input. The `interrupt` function is used in conjunction with the `Command` object to resume the graph with a value provided by the human.

```

from langgraph.types import interrupt

def human_node(state: State):
    value = interrupt(
        # Any JSON serializable value to surface to the human.
        # For example, a question or a piece of text or a set of keys in the
    state
        {
            "text_to_revise": state["some_text"]
        }
    )
    # Update the state with the human's input or route the graph based on the
    input.
    return {
        "some_text": value
    }

graph = graph_builder.compile(
    checkpoint=checkpoint # Required for `interrupt` to work
)

# Run the graph until the interrupt
thread_config = {"configurable": {"thread_id": "some_id"}}
graph.invoke(some_input, config=thread_config)

# Resume the graph with the human's input
graph.invoke(Command(resume=value_from_human), config=thread_config)

```

API Reference: [interrupt](#)

```
{'some_text': 'Edited text'}
```

Warning

Interrupts are both powerful and ergonomic. However, while they may resemble Python's `input()` function in terms of developer experience, it's important to note that they do not automatically resume execution from the interruption point. Instead, they rerun the entire node where the interrupt was used. For this reason, interrupts are typically best placed at the start of a node or in a dedicated node. Please read the [resuming from an interrupt](#) section for more details.



Full Code



Here's a full example of how to use `interrupt` in a graph, if you'd like to see the code in action.

```
from typing import TypedDict
import uuid

from langgraph.checkpoint.memory import MemorySaver
from langgraph.constants import START
from langgraph.graph import StateGraph
from langgraph.types import interrupt, Command

class State(TypedDict):
    """The graph state."""
    some_text: str

def human_node(state: State):
    value = interrupt(
        # Any JSON serializable value to surface to the human.
        # For example, a question or a piece of text or a set of keys in the
        state
    )
    return {
        # Update the state with the human's input
        "some_text": value
    }

# Build the graph
graph_builder = StateGraph(State)
# Add the human-node to the graph
graph_builder.add_node("human_node", human_node)
graph_builder.add_edge(START, "human_node")

# A checkpoint is required for `interrupt` to work.
checkpointer = MemorySaver()
graph = graph_builder.compile(
    checkpointer=checkpointer
)

# Pass a thread ID to the graph to run it.
thread_config = {"configurable": {"thread_id": uuid.uuid4()}}

# Using stream() to directly surface the `__interrupt__` information.
for chunk in graph.stream({"some_text": "Original text"},
    config=thread_config):
    print(chunk)

# Resume using Command
for chunk in graph.stream(Command(resume="Edited text"), config=thread_config):
    print(chunk)
```

API Reference: [MemorySaver](#) | [START](#) | [StateGraph](#) | [interrupt](#) | [Command](#)

```

{ '__interrupt__': (
    Interrupt(
        value={'question': 'Please revise the text', 'some_text': 'Original
text'},
        resumable=True,
        ns=['human_node:10fe492f-3688-c8c6-0d0a-ec61a43fec6'],
        when='during'
    ),
)
}
{'human_node': {'some_text': 'Edited text'}}

```

Requirements

To use `interrupt` in your graph, you need to:

1. **Specify a checkpointer** to save the graph state after each step.
2. **Call `interrupt()`** in the appropriate place. See the [Design Patterns](#) section for examples.
3. **Run the graph** with a **thread ID** until the `interrupt` is hit.
4. **Resume execution** using `invoke / ainvoke / stream / astream` (see [The Command primitive](#)).

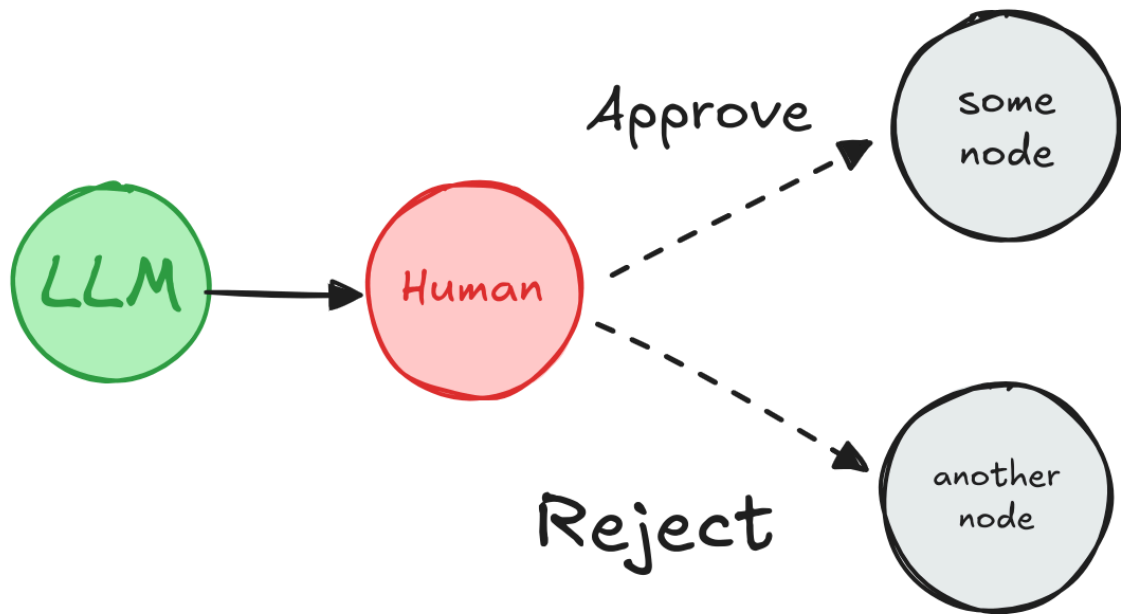
Design Patterns

There are typically three different **actions** that you can do with a human-in-the-loop workflow:

1. **Approve or Reject:** Pause the graph before a critical step, such as an API call, to review and approve the action. If the action is rejected, you can prevent the graph from executing the step, and potentially take an alternative action. This pattern often involve **routing** the graph based on the human's input.
2. **Edit Graph State:** Pause the graph to review and edit the graph state. This is useful for correcting mistakes or updating the state with additional information. This pattern often involves **updating** the state with the human's input.
3. **Get Input:** Explicitly request human input at a particular step in the graph. This is useful for collecting additional information or context to inform the agent's decision-making process or for supporting **multi-turn conversations**.

Below we show different design patterns that can be implemented using these **actions**.

Approve or Reject



Depending on the human's approval or rejection, the graph can proceed with the action or take an alternative path.

Pause the graph before a critical step, such as an API call, to review and approve the action. If the action is rejected, you can prevent the graph from executing the step, and potentially take an alternative action.

```
from typing import Literal
from langgraph.types import interrupt, Command

def human_approval(state: State) -> Command[Literal["some_node",
"another_node"]]:
    is_approved = interrupt(
        {
            "question": "Is this correct?",
            # Surface the output that should be
            # reviewed and approved by the human.
            "llm_output": state["llm_output"]
        }
    )

    if is_approved:
        return Command(goto="some_node")
    else:
        return Command(goto="another_node")

# Add the node to the graph in an appropriate location
# and connect it to the relevant nodes.
graph_builder.add_node("human_approval", human_approval)
graph = graph_builder.compile(checkpointer=checkpointer)

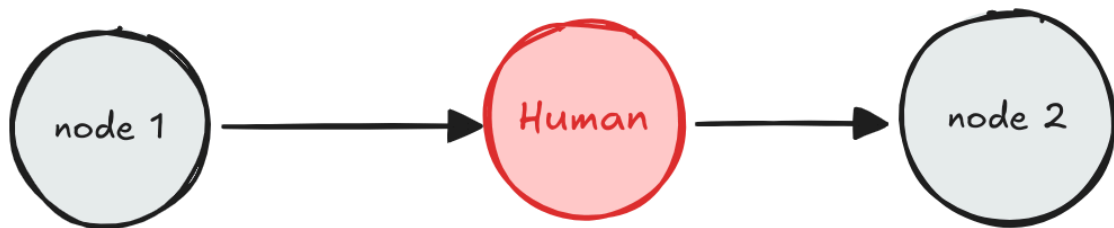
# After running the graph and hitting the interrupt, the graph will pause.
# Resume it with either an approval or rejection.
```

```
thread_config = {"configurable": {"thread_id": "some_id"}}
graph.invoke(Command(resume=True), config=thread_config)
```

API Reference: [interrupt](#) | [Command](#)

See [how to review tool calls](#) for a more detailed example.

Review & Edit State



A human can review and edit the state of the graph. This is useful for correcting mistakes or updating the state with additional information.

```
from langgraph.types import interrupt

def human_editing(state: State):
    ...
    result = interrupt(
        # Interrupt information to surface to the client.
        # Can be any JSON serializable value.
        {
            "task": "Review the output from the LLM and make any necessary
            edits.",
            "llm_generated_summary": state["llm_generated_summary"]
        }
    )

    # Update the state with the edited text
    return {
        "llm_generated_summary": result["edited_text"]
    }

# Add the node to the graph in an appropriate location
# and connect it to the relevant nodes.
graph_builder.add_node("human_editing", human_editing)
graph = graph_builder.compile(checkpointer=checkpointer)

...

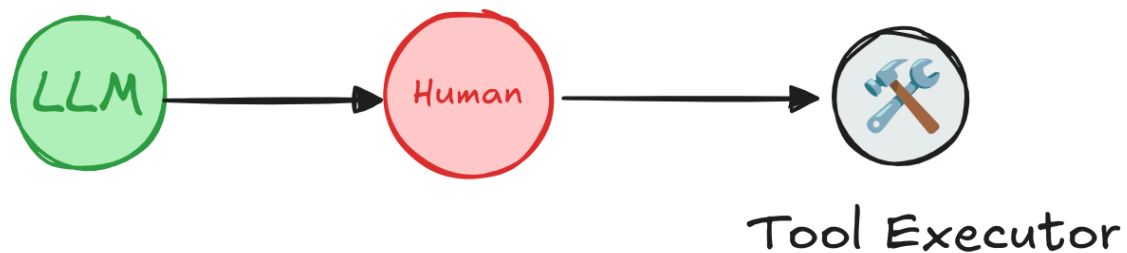
# After running the graph and hitting the interrupt, the graph will pause.
# Resume it with the edited text.
thread_config = {"configurable": {"thread_id": "some_id"}}
graph.invoke(
    Command(resume={"edited_text": "The edited text"}),
    config=thread_config
```

)

API Reference: [interrupt](#)

See [How to wait for user input using interrupt](#) for a more detailed example.

Review Tool Calls

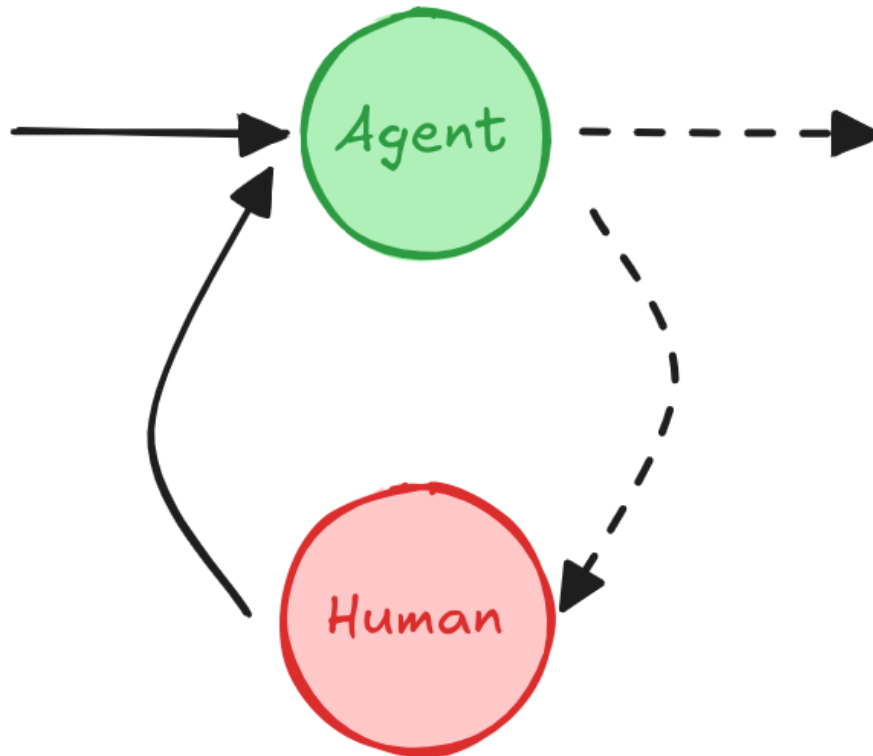


A human can review and edit the output from the LLM before proceeding. This is particularly critical in applications where the tool calls requested by the LLM may be sensitive or require human oversight.

```
def human_review_node(state) -> Command[Literal["call_llm", "run_tool"]]:  
    # This is the value we'll be providing via Command(resume=<human_review>)  
    human_review = interrupt(  
        {  
            "question": "Is this correct?",  
            # Surface tool calls for review  
            "tool_call": tool_call  
        }  
    )  
  
    review_action, review_data = human_review  
  
    # Approve the tool call and continue  
    if review_action == "continue":  
        return Command(goto="run_tool")  
  
    # Modify the tool call manually and then continue  
    elif review_action == "update":  
        ...  
        updated_msg = get_updated_msg(review_data)  
        # Remember that to modify an existing message you will need  
        # to pass the message with a matching ID.  
        return Command(goto="run_tool", update={"messages":  
[updated_message]})  
  
    # Give natural language feedback, and then pass that back to the agent  
    elif review_action == "feedback":  
        ...  
        feedback_msg = get_feedback_msg(review_data)  
        return Command(goto="call_llm", update={"messages": [feedback_msg]})
```

See [how to review tool calls](#) for a more detailed example.

Multi-turn conversation



*A **multi-turn conversation** architecture where an **agent** and **human node** cycle back and forth until the agent decides to hand off the conversation to another agent or another part of the system.*

A **multi-turn conversation** involves multiple back-and-forth interactions between an agent and a human, which can allow the agent to gather additional information from the human in a conversational manner.

This design pattern is useful in an LLM application consisting of [multiple agents](#). One or more agents may need to carry out multi-turn conversations with a human, where the human provides input or feedback at different stages of the conversation. For simplicity, the agent implementation below is illustrated as a single node, but in reality it may be part of a larger graph consisting of multiple nodes and include a conditional edge.

Using a human node per agent

In this pattern, each agent has its own human node for collecting user input. This can be achieved by either naming the human nodes with unique names (e.g., "human for agent 1",

"human for agent 2") or by using subgraphs where a subgraph contains a human node and an agent node.

```
from langgraph.types import interrupt

def human_input(state: State):
    human_message = interrupt("human_input")
    return {
        "messages": [
            {
                "role": "human",
                "content": human_message
            }
        ]
    }

def agent(state: State):
    # Agent logic
    ...

graph_builder.add_node("human_input", human_input)
graph_builder.add_edge("human_input", "agent")
graph = graph_builder.compile(checkpointer=checkpointer)

# After running the graph and hitting the interrupt, the graph will pause.
# Resume it with the human's input.
graph.invoke(
    Command(resume="hello!"),
    config=thread_config
)
```

API Reference: [interrupt](#)

Sharing human node across multiple agents

In this pattern, a single human node is used to collect user input for multiple agents. The active agent is determined from the state, so after human input is collected, the graph can route to the correct agent.

```
from langgraph.types import interrupt

def human_node(state: MessagesState) -> Command[Literal["agent_1", "agent_2", ...]]:
    """A node for collecting user input."""
    user_input = interrupt(value="Ready for user input.")

    # Determine the **active agent** from the state, so
    # we can route to the correct agent after collecting input.
    # For example, add a field to the state or use the last active agent.
    # or fill in `name` attribute of AI messages generated by the agents.
    active_agent = ...

    return Command(
```

```

        update={
            "messages": [{
                "role": "human",
                "content": user_input,
            }]
        },
        goto=active_agent,
    )

```

API Reference: [interrupt](#)

See [how to implement multi-turn conversations](#) for a more detailed example.

Validating human input

If you need to validate the input provided by the human within the graph itself (rather than on the client side), you can achieve this by using multiple interrupt calls within a single node.

```

from langgraph.types import interrupt

def human_node(state: State):
    """Human node with validation."""
    question = "What is your age?"

    while True:
        answer = interrupt(question)

        # Validate answer, if the answer isn't valid ask for input again.
        if not isinstance(answer, int) or answer < 0:
            question = f"'{answer}' is not a valid age. What is your age?"
            answer = None
            continue
        else:
            # If the answer is valid, we can proceed.
            break

    print(f"The human in the loop is {answer} years old.")
    return {
        "age": answer
    }

```

API Reference: [interrupt](#)

The Command primitive

When using the `interrupt` function, the graph will pause at the interrupt and wait for user input.

Graph execution can be resumed using the `Command` primitive which can be passed through the `invoke`, `ainvoke`, `stream` or `astream` methods.

The `Command` primitive provides several options to control and modify the graph's state during resumption:

1. **Pass a value to the `interrupt`** : Provide data, such as a user's response, to the graph using `Command(resume=value)` . Execution resumes from the beginning of the node where the `interrupt` was used, however, this time the `interrupt(...)` call will return the value passed in the `Command(resume=value)` instead of pausing the graph.

```
# Resume graph execution with the user's input.
graph.invoke(Command(resume={"age": "25"}), thread_config)
```

2. **Update the graph state**: Modify the graph state using `Command(update=update)` . Note that resumption starts from the beginning of the node where the `interrupt` was used. Execution resumes from the beginning of the node where the `interrupt` was used, but with the updated state.

```
# Update the graph state and resume.
# You must provide a `resume` value if using an `interrupt`.
graph.invoke(Command(update={"foo": "bar"}, resume="Let's go!!!"),
thread_config)
```

By leveraging `Command` , you can resume graph execution, handle user inputs, and dynamically adjust the graph's state.

Using with `invoke` and `ainvoke`

When you use `stream` or `astream` to run the graph, you will receive an `Interrupt` event that let you know the `interrupt` was triggered.

`invoke` and `ainvoke` do not return the interrupt information. To access this information, you must use the `get_state` method to retrieve the graph state after calling `invoke` or `ainvoke` .

```
# Run the graph up to the interrupt
result = graph.invoke(inputs, thread_config)
# Get the graph state to get interrupt information.
state = graph.get_state(thread_config)
# Print the state values
print(state.values)
# Print the pending tasks
print(state.tasks)
# Resume the graph with the user's input.
graph.invoke(Command(resume={"age": "25"}), thread_config)
```

```
{'foo': 'bar'} # State values
(
    PregelTask(
        id='5d8ffc92-8011-0c9b-8b59-9d3545b7e553',
        name='node_foo',
        path=('__pregel_pull', 'node_foo'),
        error=None,
        interrupts=(Interrupt(value='value_in_interrupt', resumable=True, ns=
['node_foo:5d8ffc92-8011-0c9b-8b59-9d3545b7e553'], when='during'),),
        state=None,
        result=None
    ),
) # Pending tasks. interrupts
```

How does resuming from an interrupt work?



Warning

Resuming from an `interrupt` is **different** from Python's `input()` function, where execution resumes from the exact point where the `input()` function was called.

A critical aspect of using `interrupt` is understanding how resuming works. When you resume execution after an `interrupt`, graph execution starts from the **beginning** of the **graph node** where the last `interrupt` was triggered.

All code from the beginning of the node to the `interrupt` will be re-executed.

```
counter = 0
def node(state: State):
    # All the code from the beginning of the node to the interrupt will be re-
    # executed
    # when the graph resumes.
    global counter
    counter += 1
    print(f"> Entered the node: {counter} # of times")
    # Pause the graph and wait for user input.
    answer = interrupt()
    print("The value of counter is:", counter)
    ...
```

Upon **resuming** the graph, the counter will be incremented a second time, resulting in the following output:

```
> Entered the node: 2 # of times
The value of counter is: 2
```

Common Pitfalls

Side-effects

Place code with side effects, such as API calls, **after** the `interrupt` to avoid duplication, as these are re-triggered every time the node is resumed.

Side effects before interrupt (BAD)



This code will re-execute the API call another time when the node is resumed from the `interrupt`.

This can be problematic if the API call is not idempotent or is just expensive.

```
from langgraph.types import interrupt

def human_node(state: State):
    """Human node with validation."""
    api_call(...) # This code will be re-executed when the node is resumed.
    answer = interrupt(question)
```

API Reference: [interrupt](#)

Side effects after interrupt (OK)

```
from langgraph.types import interrupt

def human_node(state: State):
    """Human node with validation."""

    answer = interrupt(question)

    api_call(answer) # OK as it's after the interrupt
```

API Reference: [interrupt](#)

Side effects in a separate node (OK)

```
from langgraph.types import interrupt

def human_node(state: State):
    """Human node with validation."""

    answer = interrupt(question)

    return {
        "answer": answer
    }

def api_call_node(state: State):
```

```
api_call(...) # OK as it's in a separate node
```

API Reference: [interrupt](#)

Subgraphs called as functions

When invoking a subgraph [as a function](#), the **parent graph** will resume execution from the **beginning of the node** where the subgraph was invoked (and where an `interrupt` was triggered). Similarly, the **subgraph**, will resume from the **beginning of the node** where the `interrupt()` function was called.

For example,

```
def node_in_parent_graph(state: State):  
    some_code() # <-- This will re-execute when the subgraph is resumed.  
    # Invoke a subgraph as a function.  
    # The subgraph contains an `interrupt` call.  
    subgraph_result = subgraph.invoke(some_input)  
    ...
```



Example: Parent and Subgraph Execution Flow



Say we have a parent graph with 3 nodes:

Parent Graph: `node_1` → `node_2` (subgraph call) → `node_3`

And the subgraph has 3 nodes, where the second node contains an `interrupt`:

Subgraph: `sub_node_1` → `sub_node_2` (`interrupt`) → `sub_node_3`

When resuming the graph, the execution will proceed as follows:

1. **Skip** `node_1` in the parent graph (already executed, graph state was saved in snapshot).
2. **Re-execute** `node_2` in the parent graph from the start.
3. **Skip** `sub_node_1` in the subgraph (already executed, graph state was saved in snapshot).
4. **Re-execute** `sub_node_2` in the subgraph from the beginning.
5. Continue with `sub_node_3` and subsequent nodes.

Here is abbreviated example code that you can use to understand how subgraphs work with interrupts. It counts the number of times each node is entered and prints the count.

```
import uuid
from typing import TypedDict

from langgraph.graph import StateGraph
from langgraph.constants import START
from langgraph.types import interrupt, Command
from langgraph.checkpoint.memory import MemorySaver

class State(TypedDict):
    """The graph state."""
    state_counter: int

counter_node_in_subgraph = 0

def node_in_subgraph(state: State):
    """A node in the sub-graph."""
    global counter_node_in_subgraph
    counter_node_in_subgraph += 1 # This code will **NOT** run again!
    print(f"Entered `node_in_subgraph` a total of {counter_node_in_subgraph} times")

counter_human_node = 0

def human_node(state: State):
    global counter_human_node
    counter_human_node += 1 # This code will run again!
    print(f"Entered human_node in sub-graph a total of {counter_human_node} times")
    answer = interrupt("what is your name?")
    print(f"Got an answer of {answer}")
```

```

checkpointer = MemorySaver()

subgraph_builder = StateGraph(State)
subgraph_builder.add_node("some_node", node_in_subgraph)
subgraph_builder.add_node("human_node", human_node)
subgraph_builder.add_edge(START, "some_node")
subgraph_builder.add_edge("some_node", "human_node")
subgraph = subgraph_builder.compile(checkpointer=checkpointer)

counter_parent_node = 0

def parent_node(state: State):
    """This parent node will invoke the subgraph."""
    global counter_parent_node

    counter_parent_node += 1 # This code will run again on resuming!
    print(f"Entered `parent_node` a total of {counter_parent_node} times")

    # Please note that we're intentionally incrementing the state counter
    # in the graph state as well to demonstrate that the subgraph update
    # of the same key will not conflict with the parent graph (until
    subgraph_state = subgraph.invoke(state)
    return subgraph_state

builder = StateGraph(State)
builder.add_node("parent_node", parent_node)
builder.add_edge(START, "parent_node")

# A checkpointer must be enabled for interrupts to work!
checkpointer = MemorySaver()
graph = builder.compile(checkpointer=checkpointer)

config = {
    "configurable": {
        "thread_id": uuid.uuid4(),
    }
}

for chunk in graph.stream({"state_counter": 1}, config):
    print(chunk)

print('--- Resuming ---')

for chunk in graph.stream(Command(resume="35"), config):
    print(chunk)

```

API Reference: [StateGraph](#) | [START](#) | [interrupt](#) | [Command](#) | [MemorySaver](#)

This will print out

```

--- First invocation ---
In parent node: {'foo': 'bar'}
Entered `parent_node` a total of 1 times
Entered `node_in_subgraph` a total of 1 times
Entered human_node in sub-graph a total of 1 times

```



```
{'__interrupt__': (Interrupt(value='what is your name?', resumable=True, ns=
['parent_node:0b23d72f-aaba-0329-1a59-ca4f3c8bad3b', 'human_node:25df717c-cb80-
57b0-7410-44e20aac8f3c'], when='during'))}

--- Resuming ---
In parent node: {'foo': 'bar'}
Entered `parent_node` a total of 2 times
Entered human_node in sub-graph a total of 2 times
Got an answer of 35
{'parent_node': None}
```

Using multiple interrupts

Using multiple interrupts within a **single** node can be helpful for patterns like [validating human input](#). However, using multiple interrupts in the same node can lead to unexpected behavior if not handled carefully.

When a node contains multiple interrupt calls, LangGraph keeps a list of resume values specific to the task executing the node. Whenever execution resumes, it starts at the beginning of the node. For each interrupt encountered, LangGraph checks if a matching value exists in the task's resume list. Matching is **strictly index-based**, so the order of interrupt calls within the node is critical.

To avoid issues, refrain from dynamically changing the node's structure between executions. This includes adding, removing, or reordering interrupt calls, as such changes can result in mismatched indices. These problems often arise from unconventional patterns, such as mutating state via `Command(resume=..., update=SOME_STATE_MUTATION)` or relying on global variables to modify the node's structure dynamically.



Example of incorrect code



```
import uuid
from typing import TypedDict, Optional

from langgraph.graph import StateGraph
from langgraph.constants import START
from langgraph.types import interrupt, Command
from langgraph.checkpoint.memory import MemorySaver

class State(TypedDict):
    """The graph state."""

    age: Optional[str]
    name: Optional[str]

def human_node(state: State):
    if not state.get('name'):
        name = interrupt("what is your name?")
    else:
        name = "N/A"

    if not state.get('age'):
        age = interrupt("what is your age?")
    else:
        age = "N/A"

    print(f"Name: {name}. Age: {age}")

    return {
        "age": age,
        "name": name,
    }

builder = StateGraph(State)
builder.add_node("human_node", human_node)
builder.add_edge(START, "human_node")

# A checkpoint must be enabled for interrupts to work!
checkpointer = MemorySaver()
graph = builder.compile(checkpointer=checkpointer)

config = {
    "configurable": {
        "thread_id": uuid.uuid4(),
    }
}

for chunk in graph.stream({"age": None, "name": None}, config):
    print(chunk)

for chunk in graph.stream(Command(resume="John", update={"name": "foo"}),
config):
    print(chunk)
```

API Reference: [StateGraph](#) | [START](#) | [interrupt](#) | [Command](#) | [MemorySaver](#)

```
{'__interrupt__': (Interrupt(value='what is your name?', resumable=True, ns=[ 'human_node:3a007ef9-c30d-c357-1ec1-86a1a70d8fba' ], when='during'),)}  
Name: N/A. Age: John  
{'human_node': {'age': 'John', 'name': 'N/A'}}
```

Additional Resources

- **[Conceptual Guide: Persistence](#)**: Read the persistence guide for more context on replaying.
- **[How to Guides: Human-in-the-loop](#)**: Learn how to implement human-in-the-loop workflows in LangGraph.
- **[How to implement multi-turn conversations](#)**: Learn how to implement multi-turn conversations in LangGraph.