

# Workflow And Agents

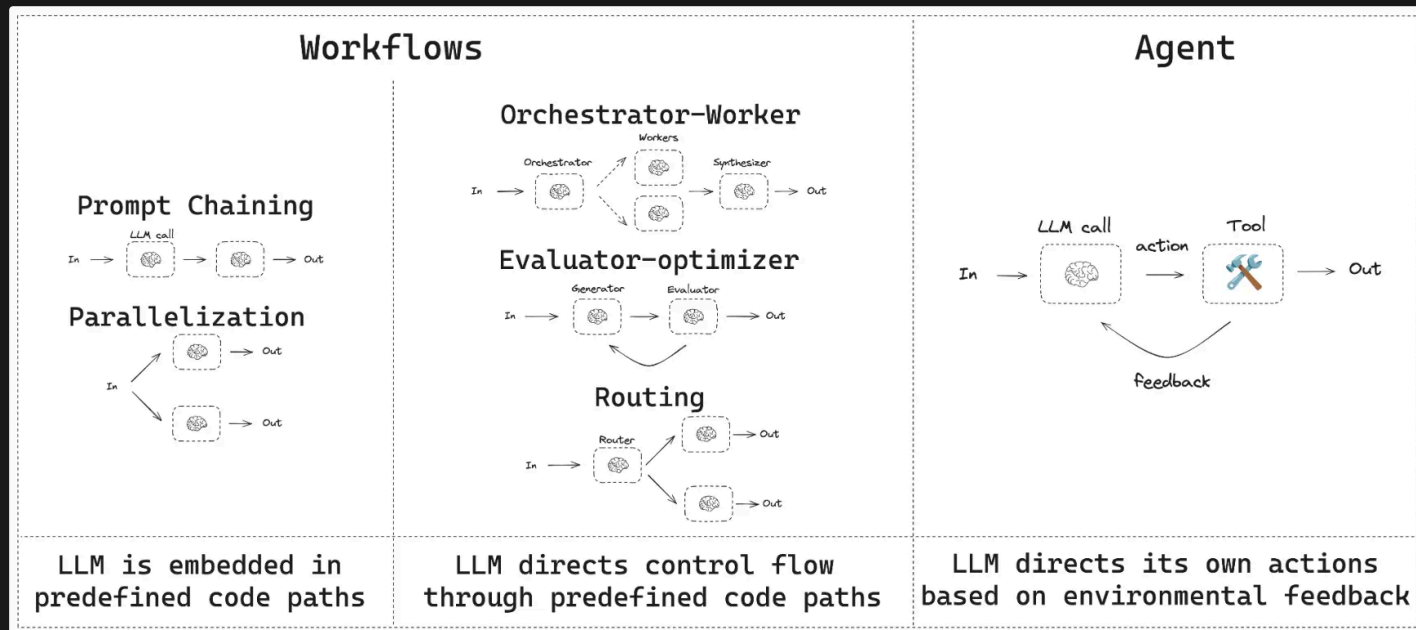
## Common Patterns

-  AnthropicAI Building effective agents

### 1. Workflow:

- Create a scaffolding of predefined code paths around LLM calls
- LLMs directs control flow through predefined code paths

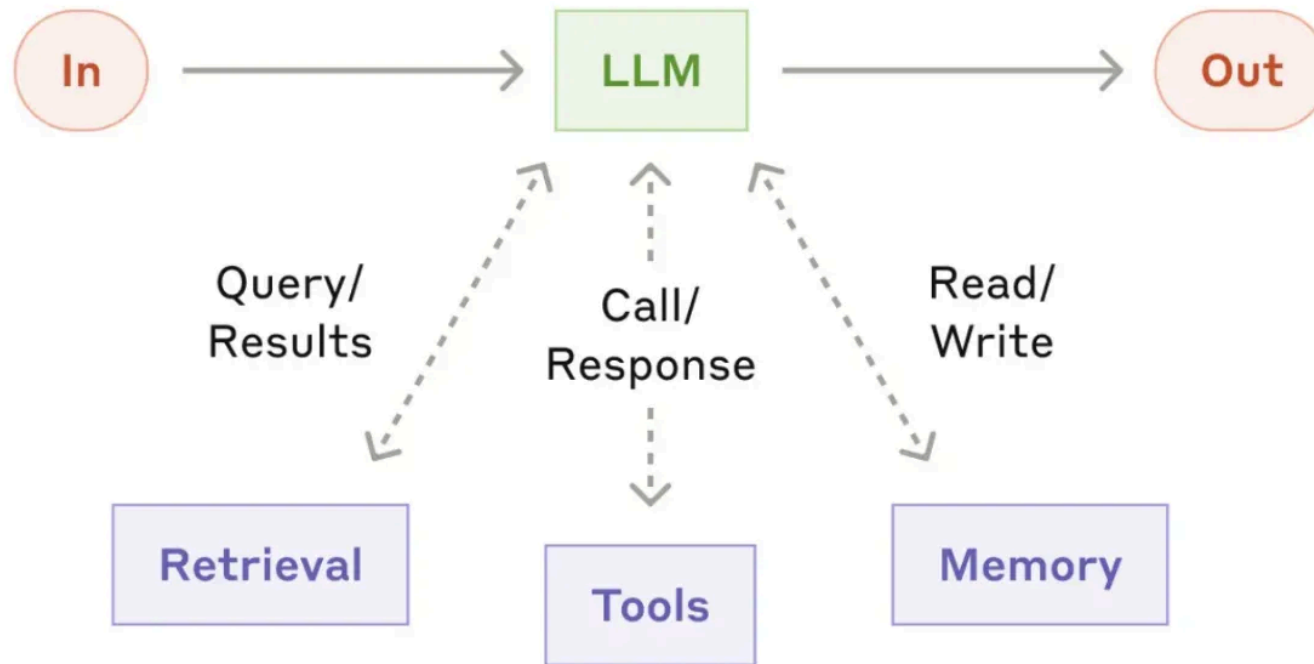
### 2. Agent: Remove this scaffolding (LLM directs its own actions, responds to feedback)



## Why Frameworks?

- Implementing these patterns *does not* require a framework like LangGraph.
- LangGraph aims to *minimize* overhead of implementing these patterns.
- LangGraph provides supporting infrastructure underneath *any*\*workflow / agent:
  - **Persistence**
    - Memory
    - Human-In-The-Loop
  - **Streaming**
    - From any LLM call or step in workflow / agent
  - **Deployment**
    - Testing, debugging, and deploying

## Augmented LLM

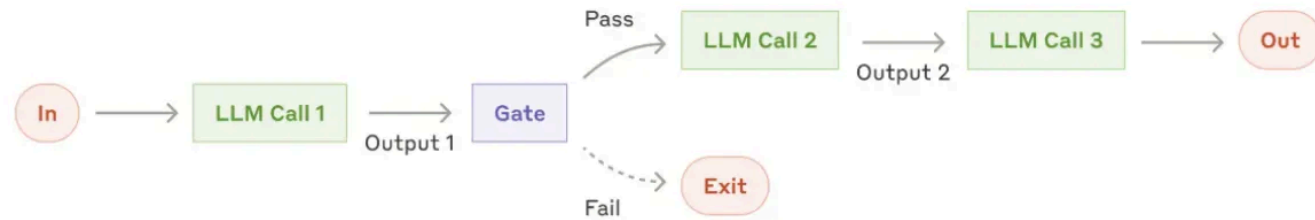


```
# LLM from langchain_anthropic import ChatAnthropic llm = ChatAnthropic(model="claude-3-5-sonnet-latest")
```

```
# Schema for structured output from pydantic import BaseModel, Field
class SearchQuery(BaseModel):
    search_query: str = Field(None, description="Query that is optimized web search.")
    justification: str = Field(None, description="Why this query is relevant to the user's request.")
# Augment the LLM with schema for structured output
structured_llm = llm.with_structured_output(SearchQuery)
# Invoke the augmented LLM
output = structured_llm.invoke("How does Calcium CT score relate to high cholesterol?")
print(output.search_query)
print(output.justification)
```

```
# Define a tool
def multiply(a: int, b: int) -> int:
    return a * b
# Augment the LLM with tools
llm_with_tools = llm.bind_tools([multiply])
# Invoke the LLM with input that triggers the tool call
msg = llm_with_tools.invoke("What is 2 times 3?")
# Get the tool call
msg.tool_calls
```

## Prompt Chaining



Each LLM call processes the output of the previous one:

- E.g., when decomposing a task into multiple LLM calls has benefit.

Example:

- Take a topic, LLM makes a joke, check the joke, improve it twice

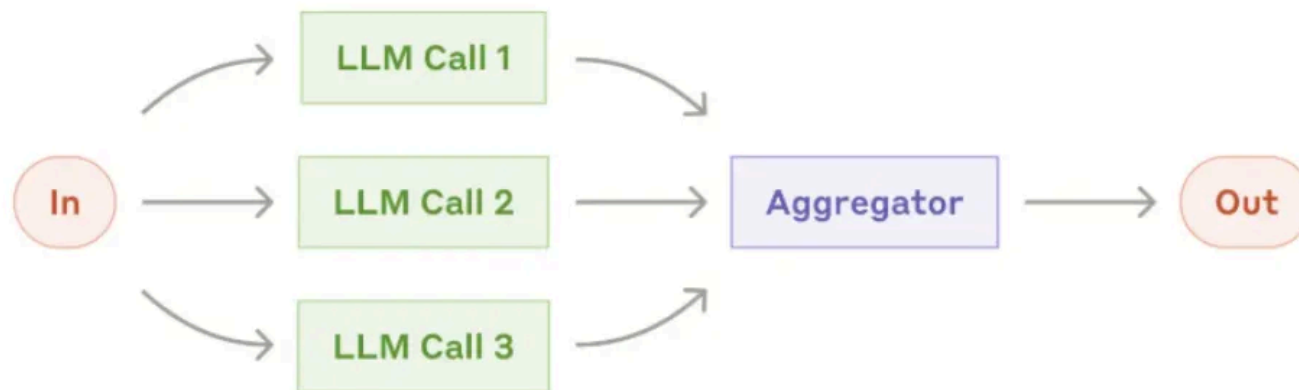
```
from typing_extensions import TypedDict # Graph state class State(TypedDict): top  
ic: str joke: str improved_joke: str final_joke: str
```

```
# Nodes
def generate_joke(state: State): """First LLM call to generate initial joke"""
    msg = llm.invoke(f"Write a short joke about {state['topic']}")
    return {"joke": msg.content}
def improve_joke(state: State): """Second LLM call to improve the joke"""
    msg = llm.invoke(f"Make this joke funnier by adding wordplay: {state['joke']}")
    return {"improved_joke": msg.content}
def polish_joke(state: State): """Third LLM call for final polish"""
    msg = llm.invoke(f"Add a surprising twist to this joke: {state['improved_joke']}")
    return {"final_joke": msg.content}
# Conditional edge function to check if the joke has a punchline
def check_punchline(state: State): """Gate function to check if the joke has a punchline"""
    # Simple check - does the joke contain "?" or "!"
    if "?" in state["joke"] or "!" in state["joke"]:
        return "Pass"
    return "Fail"
```

```
from langgraph.graph import StateGraph, START, END
from IPython.display import Image, display
# Build workflow
workflow = StateGraph(State)
# Add nodes
workflow.add_node("generate_joke", generate_joke)
workflow.add_node("improve_joke", improve_joke)
workflow.add_node("polish_joke", polish_joke)
# Add edges to connect nodes
workflow.add_edge(START, "generate_joke")
workflow.add_conditional_edges("generate_joke", check_punchline, {"Pass": "improve_joke", "Fail": END})
workflow.add_edge("improve_joke", "polish_joke")
workflow.add_edge("polish_joke", END)
# Compile chain
chain = workflow.compile()
# Show workflow
display(Image(chain.get_graph().draw_mermaid_png()))
```

```
state = chain.invoke({"topic": "cats"}) print("Initial joke:") print(state["joke"]) print("\n--- --- ---\n") if "improved_joke" in state: print("Improved joke:") print(state["improved_joke"]) print("\n--- --- ---\n") print("Final joke:") print(state["final_joke"]) else: print("Joke failed quality gate – no punchline detected!")
```

## Parallelization



- Sub-tasks can be parallelized.
  - E.g., when you want multi-perspectives for one task multi-query for RAG).
  - E.g., when independent tasks can be performed w/ different prompts.

Example:

- Take a topic, create a joke, story, and poem

```
# Graph state
class State(TypedDict):
    topic: str
    joke: str
    story: str
    poem: str
    combined_output: str
```

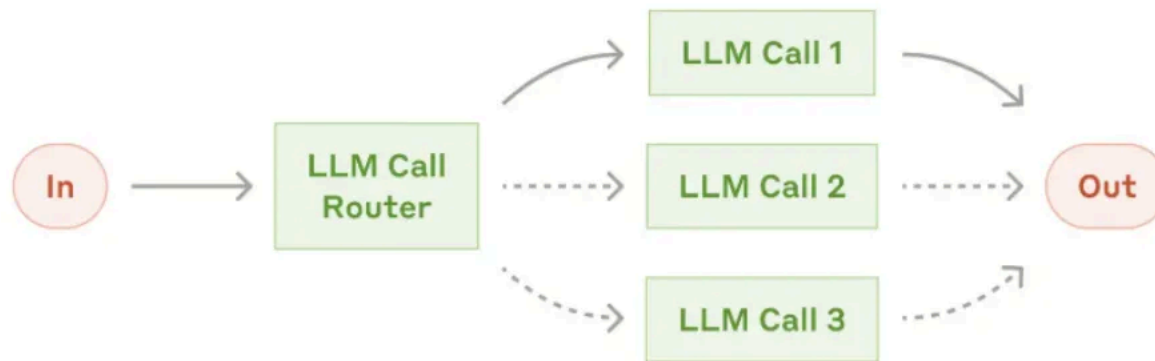
```
# Nodes
def call_llm_1(state: State):
    """First LLM call to generate initial joke"""
    msg = llm.invoke(f"Write a joke about {state['topic']}")
    return {"joke": msg.content}
def call_llm_2(state: State):
    """Second LLM call to generate story"""
    msg = llm.invoke(f"Write a story about {state['topic']}")
    return {"story": msg.content}
def call_llm_3(state: State):
    """Third LLM call to generate poem"""
    msg = llm.invoke(f"Write a poem about {state['topic']}")
    return {"poem": msg.content}
def aggregator(state: State):
    """Combine the joke and story into a single output"""
    combined = f"Here's a story, joke, and poem about {state['topic']}!\n\n"
    combined += f"STORY:\n{state['story']}\n\n"
    combined += f"JOKE:\n{state['joke']}\n\n"
    combined += f"POEM:\n{state['poem']}\n\n"
    return {"combined_output": combined}
```



```
# Build workflow parallel_builder = StateGraph(State) # Add nodes parallel_builde
r.add_node("call_llm_1", call_llm_1) parallel_builder.add_node("call_llm_2", call
_llm_2) parallel_builder.add_node("call_llm_3", call_llm_3) parallel_builder.add_
node("aggregator", aggregator) # Add edges to connect nodes parallel_builder.add_
edge(START, "call_llm_1") parallel_builder.add_edge(START, "call_llm_2") parallel
_builder.add_edge(START, "call_llm_3") parallel_builder.add_edge("call_llm_1", "a
ggregator") parallel_builder.add_edge("call_llm_2", "aggregator") parallel_builde
r.add_edge("call_llm_3", "aggregator") parallel_builder.add_edge("aggregator", EN
D) parallel_workflow = parallel_builder.compile() # Show workflow display(Image(p
arallel_workflow.get_graph().draw_mermaid_png()))
```

```
state = parallel_workflow.invoke({"topic": "cats"}) print(state["combined_outpu
t"])
```

## Routing



Routing classifies an input and directs it to a specialized followup task.

- E.g., when routing a question to different retrieval systems.

Example:

- Route an input between joke, story, and poem

```
from typing_extensions import Literal # Schema for structured output to use as routing logic
class Route(BaseModel):
    step: Literal["poem", "story", "joke"] = Field(
        None, description="The next step in the routing process"
    ) # Augment the LLM with schema for structured output
router = llm.with_structured_output(Route)
```

```
# State class State(TypedDict): input: str decision: str output: str
```

```
from langchain_core.messages import HumanMessage, SystemMessage # Nodes
def llm_call_1(state: State): """Write a story""" print("Write a story") result = llm.invoke(state["input"]) return {"output": result.content}
def llm_call_2(state: State): """Write a joke""" print("Write a joke") result = llm.invoke(state["input"]) return {"output": result.content}
def llm_call_3(state: State): """Write a poem""" print("Write a poem") result = llm.invoke(state["input"]) return {"output": result.content}
def llm_call_router(state: State): """Route the input to the appropriate node""" # Run the augmented LLM with structured output to serve as routing logic
decision = router.invoke([SystemMessage(content="Route the input to story, joke, or poem based on the user's request."), HumanMessage(content=state["input"])])
return {"decision": decision.step} # Conditional edge function to route to the appropriate node
def route_decision(state: State): # Return the node name you want to visit next
if state["decision"] == "story": return "llm_call_1"
elif state["decision"] == "joke": return "llm_call_2"
elif state["decision"] == "poem": return "llm_call_3"
```

```
# Build workflow
router_builder = StateGraph(State) # Add nodes
router_builder.add_node("llm_call_1", llm_call_1)
router_builder.add_node("llm_call_2", llm_call_2)
router_builder.add_node("llm_call_3", llm_call_3)
router_builder.add_node("llm_call_router", llm_call_router) # Add edges to connect nodes
router_builder.add_edge(START, "llm_call_router")
router_builder.add_conditional_edges("llm_call_router", route_decision, { # Name returned by route_decision : Name of next node to visit
    "llm_call_1": "llm_call_1", "llm_call_2": "llm_call_2", "llm_call_3": "llm_call_3", }, )
router_builder.add_edge("llm_call_1", END)
router_builder.add_edge("llm_call_2", END)
router_builder.add_edge("llm_call_3", END) # Compile workflow
router_workflow = router_builder.compile() # Show the workflow
display(Image(router_workflow.get_graph().draw_mermaid_png()))
```

```
state = router_workflow.invoke({"input": "Write me a joke about cats"})
print(state["output"])
```

## Orchestrator-Worker



Orchestrator breaks down a task and delegates each sub-task to workers.

- E.g., planning a report where LLM can determine the number of sections.

Example

- Take a topic, plan a report of section, have each worker write a section

```
from typing import Annotated, List
import operator # Schema for structured output
to use in planning
class Section(BaseModel):
    name: str = Field(
        description="Name for this section of the report.",
    )
    description: str = Field(
        description="Brief overview of the main topics and concepts to be covered in this section.",
    )
class Sections(BaseModel):
    sections: List[Section] = Field(
        description="Sections of the report.",
    )
# Augment the LLM with schema for structured output
planner = llm.with_structured_output(Sections)
```

```
# Graph state
class State(TypedDict):
    topic: str # Report topic
    sections: list[Section] # List of report sections
    completed_sections: Annotated[list, operator.add] # All workers write to this key in parallel
    final_report: str # Final report

# Worker state
class WorkerState(TypedDict):
    section: Section
    completed_sections: Annotated[list, operator.add]
```

```
# Nodes
def orchestrator(state: State):
    """Orchestrator that generates a plan for the report"""
    # Generate queries
    report_sections = planner.invoke([
        SystemMessage(content="Generate a plan for the report."),
        HumanMessage(content=f"Here is the report topic: {state['topic']}"),
    ])
    return {"sections": report_sections.sections}

def llm_call(state: WorkerState):
    """Worker writes a section of the report"""
    # Generate section
    section = llm.invoke([
        SystemMessage(content="Write a report section."),
        HumanMessage(content=f"Here is the section name: {state['section'].name} and description: {state['section'].description}"),
    ])
    # Write the updated section to completed sections
    return {"completed_sections": [section.content]}

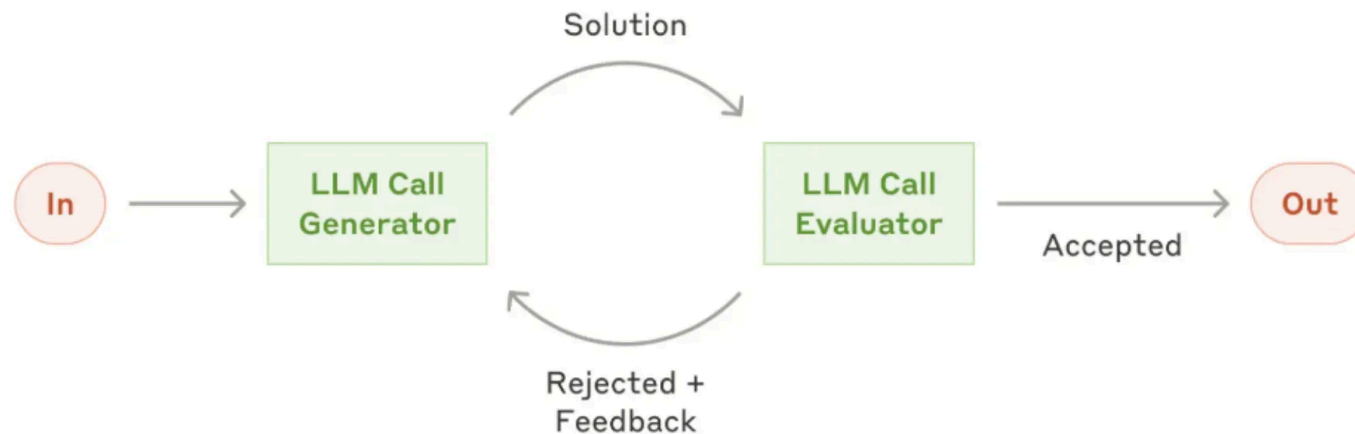
def synthesizer(state: State):
    """Synthesize full report from sections"""
    # List of completed sections
    completed_sections = state["completed_sections"]
    # Format completed section to str to use as context for final sections
    completed_report_sections = "\n\n---\n\n".join(completed_sections)
    return {"final_report": completed_report_sections}

# Conditional edge function to create llm_call workers that each write a section of the report
def assign_workers(state: State):
    """Assign a worker to each section in the plan"""
    # Kick off section writing in parallel via Send() API
    return [Send("llm_call", {"section": s}) for s in state["sections"]]
```

```
from langgraph.constants import Send # Build workflow orchestrator_worker_builder
= StateGraph(State) # Add the nodes orchestrator_worker_builder.add_node("orchestrator", orchestrator)
orchestrator_worker_builder.add_node("llm_call", llm_call)
orchestrator_worker_builder.add_node("synthesizer", synthesizer) # Add edges to connect nodes
orchestrator_worker_builder.add_edge(START, "orchestrator")
orchestrator_worker_builder.add_conditional_edges("orchestrator", assign_workers, ["llm_call"])
orchestrator_worker_builder.add_edge("llm_call", "synthesizer")
orchestrator_worker_builder.add_edge("synthesizer", END) # Compile the workflow
orchestrator_worker = orchestrator_worker_builder.compile() # Show the workflow
display(Image(orchestrator_worker.get_graph().draw_mermaid_png()))
```

```
state = orchestrator_worker.invoke({"topic": "Create a report on LLM scaling laws"})
from IPython.display import Markdown
Markdown(state["final_report"])
```

## Evaluator-optimizer



One LLM call generates a response while another provides evaluation and feedback in a loop.

- E.g., when grading the quality of responses from a RAG system (for hallucinations).

```
# Schema for structured output to use in evaluation
class Feedback(BaseModel):
    grade: Literal["funny", "not funny"] = Field(
        description="Decide if the joke is funny or not.",
    )
    feedback: str = Field(
        description="If the joke is not funny, provide feedback on how to improve it.",
    )
# Augment the LLM with schema for structured output
evaluator = llm.with_structured_output(Feedback)
```

```
# Graph state
class State(TypedDict):
    joke: str
    topic: str
    feedback: str
    funny_or_not: str
```



```
# Nodes
def llm_call_generator(state: State): """LLM generates a joke""" if state.get("feedback"): msg = llm.invoke( f"Write a joke about {state['topic']} but take into account the feedback: {state['feedback']}" ) else: msg = llm.invoke(f"Write a joke about {state['topic']}") return {"joke": msg.content}
def llm_call_evaluator(state: State): """LLM evaluates the joke""" grade = evaluator.invoke(f"Grade the joke {state['joke']}") return {"funny_or_not": grade.grade, "feedback": grade.feedback}
# Conditional edge function to route back to joke generator or end based upon feedback from the evaluator
def route_joke(state: State): """Route back to joke generator or end based upon feedback from the evaluator""" if state["funny_or_not"] == "funny": return "Accepted" elif state["funny_or_not"] == "not funny": return "Rejected + Feedback"
```

```
# Build workflow
optimizer_builder = StateGraph(State)
# Add the nodes
optimizer_builder.add_node("llm_call_generator", llm_call_generator)
optimizer_builder.add_node("llm_call_evaluator", llm_call_evaluator)
# Add edges to connect nodes
optimizer_builder.add_edge(START, "llm_call_generator")
optimizer_builder.add_edge("llm_call_generator", "llm_call_evaluator")
optimizer_builder.add_conditional_edges("llm_call_evaluator", route_joke, { # Name returned by route_joke : Name of next
```