Building Effective Agents with LangGraph

LangChain
120K subscribers

Subscribe

3.4K | Share | Clip | Save

111,728 views Jan 27, 2025
Anthropic's recent blog post on "Building Effective Agents" lays out the difference between "agents" and "workflows", and presents a number of common patterns for both. Here, we implement every workflow and agent pattern covered in the blog from scratch using LangGraph. We explain the key differences between workflows and agents, when to use each approach, and how to implement them effectively. We also cover the benefits you can gain from using LangGraph as a framework.

Documentation:
https://langchain-ai.github.io/langgr...

Video notes:
https://mirror-feeling-d80.notion.sit...

Timestamps:
0:00 Introduction & Key Concepts
1:00 Understanding Workflows vs Agents
2:00 Why Use Frameworks? Benefits of LangGraph
4:00 Building Block: Augmented LLM
5:00 Pattern 1: Basic Prompt Chaining
9:00 Pattern 2: Parallelization
11:00 Pattern 3: Routing with LLMs
14:00 Pattern 4: Orchestrator-Worker Pattern
20:00 Pattern 5: Evaluator-Optimizer Workflow
24:00 Building Agents: Beyond Workflows
27:00 Implementing a Basic Agent Loop
30:00 Conclusion & LangGraph Benefits

Home

Memory
Human-in-the-loop
Multi-agent
Evals
Deployment
UI

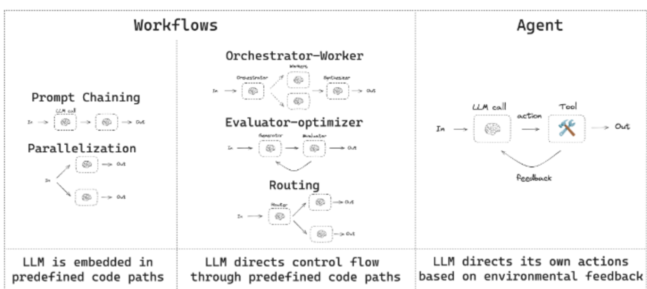
LangGraph framework
Agent architectures
Overview
Workflows & agents
Graphs
Streaming
Persistence
Memory
Human-in-the-loop
Breakpoints
Time travel
Tools
Subgraphs
Multi-agent

Workflows and Agents

This guide reviews common patterns for agentic systems. In describing these systems, it can be useful to make a distinction between "workflows" and "agents". One way to think about this difference is nicely explained in Anthropic's `Building Effective Agents` blog post:

> Workflows are systems where LLMs and tools are orchestrated through predefined code paths. Agents, on the other hand, are systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Here is a simple way to visualize these differences:

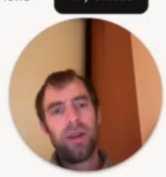


ANTHROP\C   Claude | Research | Company | Careers | News | Try Claude

Product

Building effective agents

Dec 19, 2024



Over the past year, we've worked with dozens of teams building large language model (LLM) agents across industries. Consistently, the most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns.

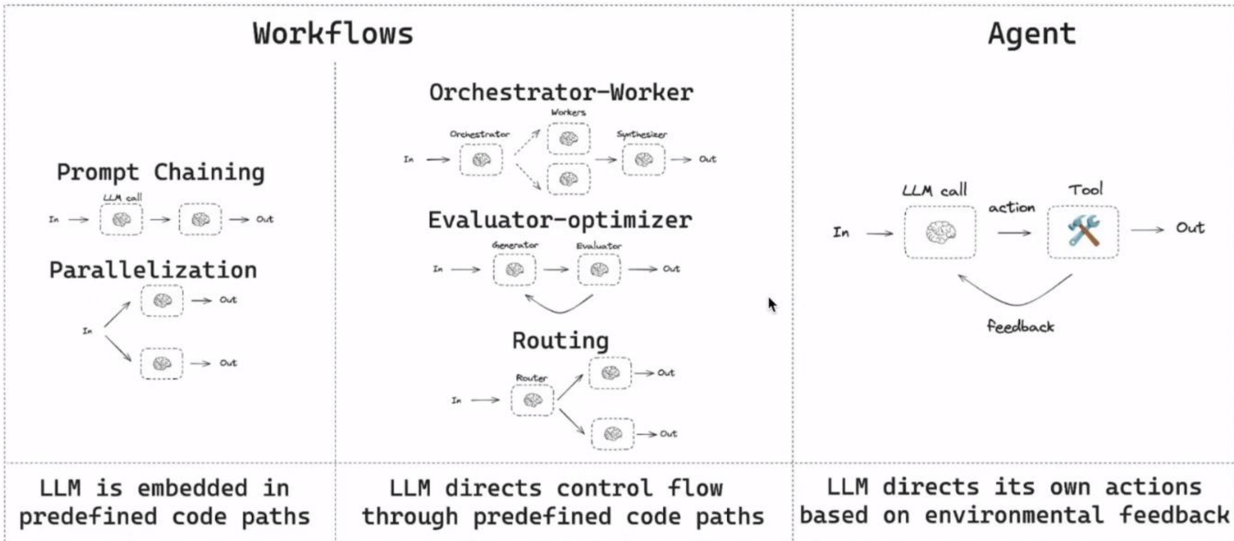
In this post, we share what we've learned from working with our customers and building agents ourselves, and give practical advice for developers on building effective agents.

What are agents?

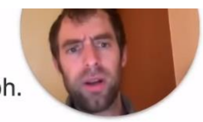
"Agent" can be defined in several ways. Some customers define agents as fully autonomous systems that operate independently over extended periods, using various tools to accomplish complex tasks. Others use the term to describe more prescriptive implementations that follow predefined workflows. At Anthropic, we categorize all

We will be building the workflows and agents here from scratch

1. **Workflow:**
   a. Create a scaffolding of predefined code paths around LLM calls
   b. LLMs directs control flow through predefined code paths
2. **Agent**: Remove this scaffolding (LLM directs its own **actions**, responds to **feedback**)



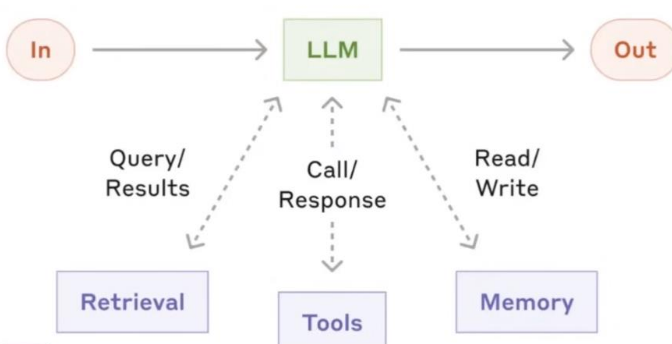
Agents remove the workflow scaffolding/bounding and makes the decisions by themselves.



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

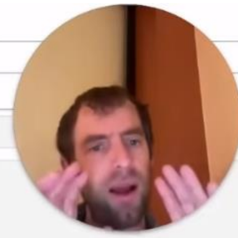
Augmented LLM



jupyter workflows_agents_scratch Last Checkpoint: 2 days ago

File Edit View Run Kernel Settings Help

Code



[ ]:
```
! pip install langchain_core langchain-anthropic langgraph
```

[ ]:
```python
import os, getpass


def _set_env(var: str):
    if not os.environ.get(var):
        os.environ[var] = getpass.getpass(f"{var}: ")


_set_env("ANTHROPIC_API_KEY")
```

[8]:
```python
# LLM
from langchain_anthropic import ChatAnthropic
llm = ChatAnthropic(model="claude-3-5-sonnet-latest")
```

```python
llm = ChatAnthropic(model="claude-3-5-sonnet-latest")
```

[9]:
```python
# 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, justification="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)
```

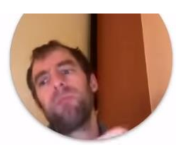
```
relationship between Calcium CT score and high cholesterol cardiovascular risk
This query will help understand the connection between coronary calcium scoring and cholesterol le
sk.
```

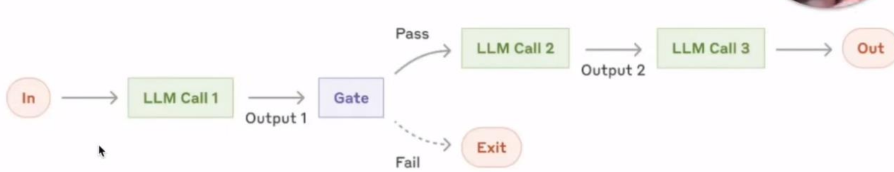

```
sk.
```


[10]:
```python
# 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
```

[10]:
```
[{'name': 'multiply',
  'args': {'a': 2, 'b': 3},
  'id': 'toolu_013EhX1LuBjqv2XyZfjosFqV',
  'type': 'tool_call'}]
```

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

```python
from typing_extensions import TypedDict

# Graph state
class State(TypedDict):
    topic: str
    joke: str
    improved_joke: str
    final_joke: str
```

```python
# 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"
```

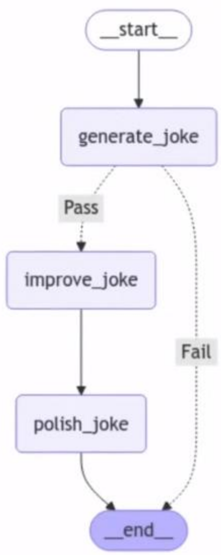
LangGraph allows us to use a container or **state** that we can use to hold values and pass to LLMs in the steps for populating as above. The **Gate** logic is implemented using the checks in the **check_punchline()** function with the result being strings of **"Pass"** or **"Fail"** in LangGraph

```python
# 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"
```

```python
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 = workflow.compile()

# Show workflow
display(Image(chain.get_graph().draw_mermaid_png()))
```

We then lay out the workflow in LangGraph as above

```
# 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!")
```



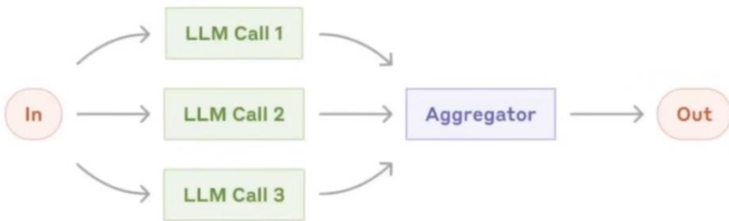
```
    print("Joke failed quality gate - no punchline detected!")
```



```
Initial joke:
Here's a short cat joke:

What do you call a cat that likes to go bowling?

An alley cat! 🐱🎳

--- --- ---

Improved joke:
Here's the joke with added wordplay:

What do you call a cat that likes to go bowling?

An alley cat! They're always purr-fect at getting strikes and never kitten around when it's time t

(The added wordplay includes "purr-fect" instead of "perfect" and "kitten around" instead of "kidd
g terms like "strikes" and "spare")

--- --- ---

Final joke:
Here's the joke with a surprising twist:

What do you call a cat that likes to go bowling?

An alley cat! They're always purr-fect at getting strikes and never kitten around when it's time t
just been knocking over pins with their laser pointer this whole time! 🐱🎳

(The twist reveals that the cat hasn't actually been bowling at all, but rather playing with a las
usly distracted by - which plays on the audience's expectations while staying true to typical cat
```

We can now see the output of the invoked workflow above and the results from each step.

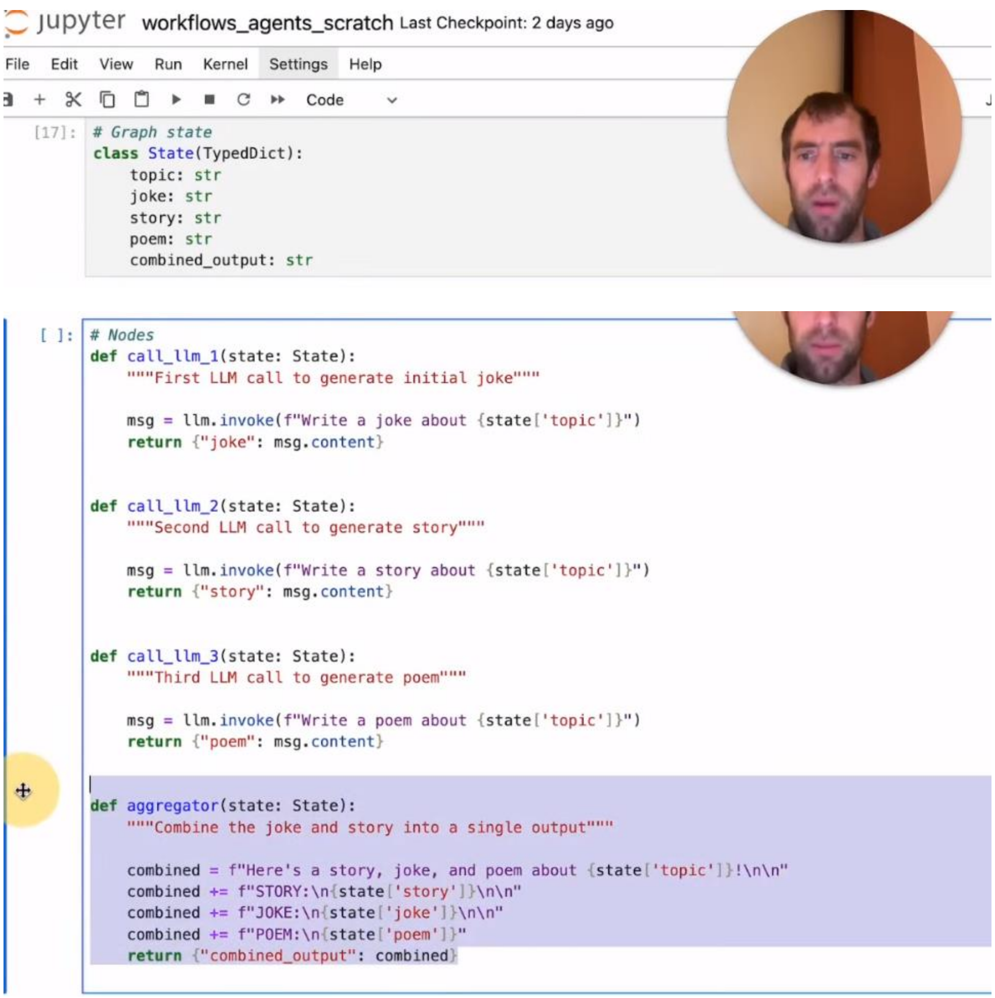
Parallelization



- 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



jupyter workflows_agents_scratch Last Checkpoint: 2 days ago

File Edit View Run Kernel Settings Help

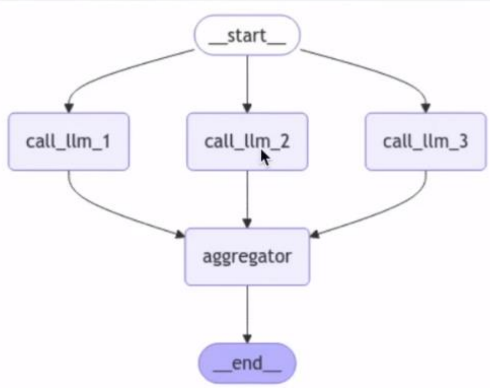
[17]:
```python
# Graph state
class State(TypedDict):
    topic: str
    joke: str
    story: str
    poem: str
    combined_output: str
```

[ ]:
```python
# 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']}"
    return {"combined_output": combined}
```

```python
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']}"
    return {"combined_output": combined}
```

[ ]:
```python
# Build workflow
parallel_builder = StateGraph(State)

# Add nodes
parallel_builder.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", "aggregator")
parallel_builder.add_edge("call_llm_2", "aggregator")
parallel_builder.add_edge("call_llm_3", "aggregator")
parallel_builder.add_edge("aggregator", END)
parallel_workflow = parallel_builder.compile()

# Show workflow
display(Image(parallel_workflow.get_graph().draw_mermaid_png()))
```

```python
# Show workflow
display(Image(parallel_workflow.get_graph().draw_mermaid_png()))
```



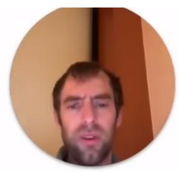
[*]:
```python
state = parallel_workflow.invoke({"topic": "cats"})
print(state["combined_output"])
```

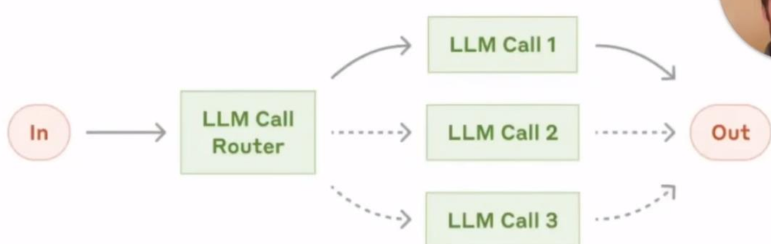
[20]:
```python
state = parallel_workflow.invoke({"topic": "cats"})
print(state["combined_output"])
```

```
Here's a story, joke, and poem about cats!

STORY:
Here's a short story about cats:

The Unlikely Friends

Luna was a sleek black cat who lived in a cozy house at the end of Maple Street. She spent most of
lls and watching the world outside with her bright yellow eyes. While other cats in the neighborho
ed her solitude—until one rainy afternoon changed everything.

A scrawny orange tabby appeared in her garden, soaking wet and shivering. Luna watched from her wa
ter under her favorite rose bush. Something about the cat's pitiful state stirred something in Lun

Against her better judgment, she meowed loudly until her human noticed the visitor. Soon enough, t
—was brought inside, dried off, and given a warm meal.

Luna initially kept her distance, observing the newcomer from around corners and behind furniture.
his friendly attempts to connect. He would leave small toys near her favorite spots and playfully

Gradually, Luna's icy demeanor began to thaw. She found herself enjoying Oliver's silly antics and
ong, the two cats became inseparable—napping together in patches of sunlight, sharing treats, and

Their humans were amazed at the transformation. The once-solitary Luna had found a best friend in
etimes the most unexpected friendships are the best ones.

Now, visitors to the house at the end of Maple Street would always find two cats curled up togethe
sunset—contentedly purring in their shared home.

The End.
```

```
The End.

JOKE:
Here's a cat joke for you:

Why don't cats like online shopping?

They prefer a cat-alog!

POEM:
Here's a poem about cats:

Soft paws and whiskers long,
Graceful hunters, proud and strong.
Eyes that glow in darkest night,
Watching, stalking with delight.

Purring warmth upon my lap,
Curled up tight for afternoon nap.
Independent, yet so sweet,
Landing always on their feet.

Masters of the window sill,
Mysterious creatures of their own will.
Playful one moment, aloof the next,
By their own rules, they're never vexed.

Velvet fur and gentle grace,
Each whisker perfectly in place.
Faithful friends through joy and strife,
Adding magic to our life.
```

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

Routing

```python
[21]: 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)
```

```python
[23]: # State
class State(TypedDict):
    input: str
    decision: str
    output: str
```

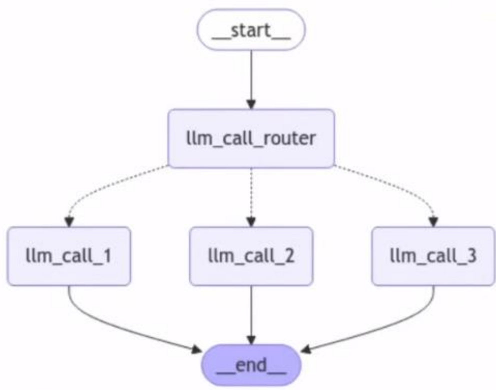
We can take an LLM and give it a structured output, this guarantees that the LLM will produce a structured object in the format we specified. E.g. as a “poem”, “story”, “joke” for above example.

```python
from langchain_core.messages import HumanMessage, SystemMessage

# Nodes
def llm_call_1(state: State):
    """Write a story"""

    result = llm.invoke(state["input"])
    return {"output": result.content}


def llm_call_2(state: State):
    """Write a joke"""

    result = llm.invoke(state["input"])
    return {"output": result.content}


def llm_call_3(state: State):
    """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"
```

```python
# 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",
```

```python
    "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()))
```

```python
# Show the workflow
display(Image(router_workflow.get_graph().draw_mermaid_png()))
```




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

```
Write a joke
Here's a cat joke for you:

Why don't cats like online shopping?
They prefer a cat-alog!
```

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

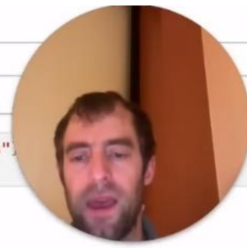
The Orchestrator LLM here selects the appropriate LLMs based on its reasoning at runtime only, a deep research agent.

```python
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.",
    )
```

jupyter workflows_agents_scratch Last Checkpoint: 8 minutes ago

File Edit View Run Kernel Settings Help

Code



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

```
Write a joke
Here's a cat joke for you:

Why don't cats like online shopping?
They prefer a cat-alog!
```

Orchestrator Worker

```python
[ ]: 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)
```

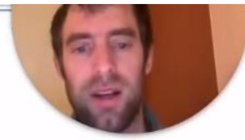
The orchestrator/**planner** is going to take an **input**, **reflect** on it to produce a list of **Sections** at runtime.

```python
# Augment the LLM with schema for structured output
planner = llm.with_structured_output(Sections)
```

```python
[ ]: # 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]
```

We create a **State** for the orchestrator graph that will contain a **topic** from a user, a list of **sections**, a list of **completed_sections** for the workers to operate on and a **final_report**. Each worker will also have the state as **WorkerState** since they are self-contained objects with independent input/**section** and output/**completed_output**.

```
[ ]: # 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: {stat
            ),
        ]
    )

    # Write the updated section to completed sections
    return {"completed_sections": [section.content]}


def synthesizer(state: State):
    """Synthesize full report from sections"""
```



```
(content="Write a report section."),

'Here is the section name: {state['section'].name} and description: {state['section'].description}"
```

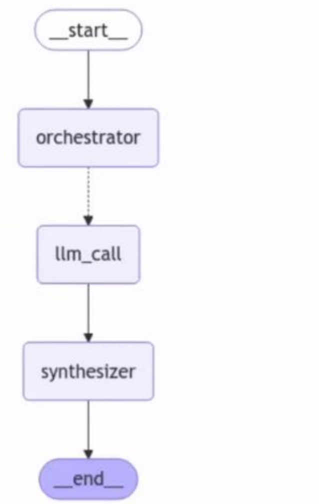
```
def synthesizer(state: State):
    """Synthesize full report from sections"""

    # 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"]]
```

The **Send()** function is used to dynamically spurn the workers as needed for each defined state section.

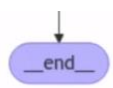
```python
    # Kick off section writing in parallel via Send() API
    return [Send("llm_call", {"section": s}) for s in state["sections"]]
```

```python
# Show the workflow
display(Image(orchestrator_worker.
```

```python
[ ]: 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()))
```



The dotted arrow line tells you that the orchestrator is going to be spurning these workers that will make the **llm_call()** dynamically based on the orchestrator's plan.




```python
[ ]: state = orchestrator_worker.invoke({"topic": "Create a report on LLM scaling laws"})

[34]: state["sections"]

[34]: [Section(name='Introduction to LLM Scaling Laws', description='Overview of what scaling laws are i
ls, their importance in AI research, and historical background'),
 Section(name='Fundamental Scaling Relationships', description='Detailed examination of key scalin
rameter scaling, and dataset size scaling, with mathematical foundations and empirical observation
 Section(name='Chinchilla Scaling Laws', description="Analysis of DeepMind's Chinchilla paper find
n, and the relationship between model size and training tokens"),
 Section(name='Impact on Model Architecture', description='How scaling laws influence model archit
width trade-offs, attention mechanisms, and computational efficiency'),
 Section(name='Economic and Computational Implications', description='Discussion of training costs
onomic considerations in scaling LLMs according to established laws'),
 Section(name='Limitations and Challenges', description='Examination of current limitations in sca
s improvements, and challenges in extending current scaling patterns'),
 Section(name='Future Directions', description='Exploration of potential future developments in sc
ling comparisons, alternative scaling approaches, and sustainability considerations')]

•[33]: from IPython.display import Markdown
Markdown(state["final_report"])
```

```python
•[33]: from IPython.display import Markdown
Markdown(state["final_report"])
```

Introduction to LLM Scaling Laws

Scaling laws for Large Language Models (LLMs) represent fundamental empirical relationships that describe how model
key parameters such as model size, dataset size, and compute resources. These mathematical relationships have becom
predicting the behavior of language models as they grow in scale, helping researchers and organizations make informed
resource allocation.

The concept of scaling laws emerged from seminal research by OpenAI and others in the late 2010s, which demonstrate
follows surprisingly predictable patterns as models increase in size. These patterns typically follow power-law relationshi
loss or perplexity improve smoothly as a function of compute, parameters, or data, often appearing as straight lines whe

Key aspects of scaling laws include:

Fundamental Scaling Relationships ¶

Introduction

The field of artificial intelligence has revealed several fundamental scaling relationships that govern how model performa
parameters, and data. Understanding these scaling laws is crucial for predicting model capabilities, making informed arc
allocating resources in AI development.

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).

Evaluator - Optimizer

[35]:
```python
# 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)
```

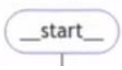
[36]:
```python
# Graph state
class State(TypedDict):
    joke: str
    topic: str
    feedback: str
    funny_or_not: str
```

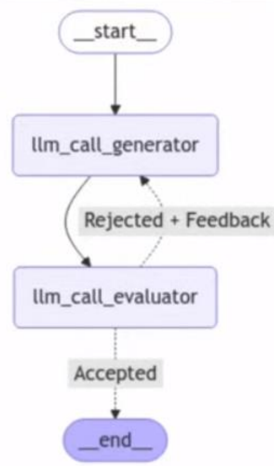

```python
    funny_or_not: str
```


[ ]:
```python
# 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['feed
        )
    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 ev
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"
```

```
    elif state["funny_or_not"] == "not funny":
        return "Rejected + Feedback"
```

```
[38]: # 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 node to visit
        "Accepted": END,
        "Rejected + Feedback": "llm_call_generator",
    },
)

# Compile the workflow
optimizer_workflow = optimizer_builder.compile()

# Show the workflow
display(Image(optimizer_workflow.get_graph().draw_mermaid_png()))
```






```
[ ]: state = optimizer_workflow.invoke({"topic": "Cats"})
print(state["joke"])
```

```
[39]: state = optimizer_workflow.invoke({"topic": "Cats"})
print(state["joke"])
```

```
Here's a cat joke for you:

Why don't cats like online shopping?

They prefer a cat-alog!

*ba dum tss* 😸
```

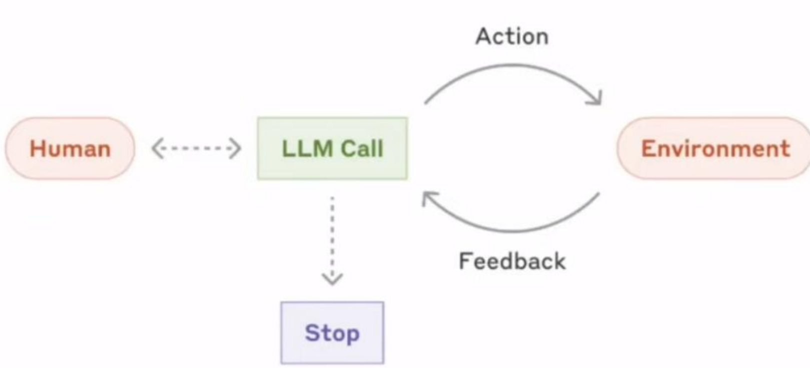
```
[40]: print(state["feedback"])
```

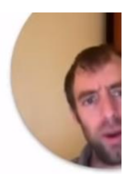
```
This is a clever play on words using "catalog" and "cat," making it a solid pun that's both releva
s. The setup is clear and the punchline delivers well. No significant improvement needed as it wor
joke that cat lovers would appreciate.
```

```
[41]: print(state["funny_or_not"])
```

```
funny
```

Agent



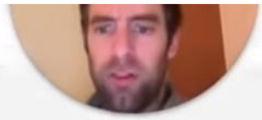


- Agents plan, take actions (via tool-calling), and respond to feedback (in a loop).
  - E.g., when solving open-ended problems that you cannot lay out as a workflow

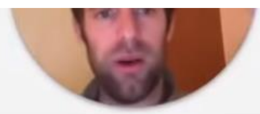
Agent

[42]:

```python
from langchain_core.tools import tool

# Define tools
@tool
def multiply(a: int, b: int) -> int:
    """Multiply a and b.

    Args:
        a: first int
        b: second int
    """
    return a * b


@tool
def add(a: int, b: int) -> int:
    """Adds a and b.

    Args:
        a: first int
        b: second int
    """
    return a + b


@tool
def divide(a: int, b: int) -> float:
    """Divide a and b.

    Args:
        a: first int
        b: second int
    """
    return a / b
```



```python
# Augment the LLM with tools
tools = [add, multiply, divide]
tools_by_name = {tool.name: tool for tool in tools}
llm_with_tools = llm.bind_tools(tools)
```

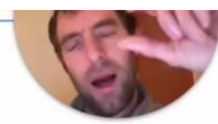


[ ]:

```python
from langgraph.graph import MessagesState
```

[ ]:

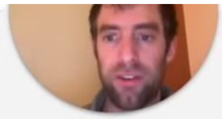
```python
from langgraph.graph import MessagesState
from langchain_core.messages import ToolMessage

# Nodes
def llm_call(state: MessagesState):
    """LLM decides whether to call a tool or not"""

    return {
        "messages": [
            llm_with_tools.invoke(
                [
                    SystemMessage(
                        content="You are a helpful assistant tasked with performing arithmetic on a set of inputs."
                    )
                ]
                + state["messages"]
            )
        ]
    }


def tool_node(state: dict):
    """Performs the tool call"""

    result = []
    for tool_call in state["messages"][-1].tool_calls:
        tool = tools_by_name[tool_call["name"]]
        observation = tool.invoke(tool_call["args"])
        result.append(ToolMessage(content=observation, tool_call_id=tool_call["id"]))
    return {"messages": result}


# Conditional edge function to route to the tool node or end based upon whether the LLM made a tool call
def should_continue(state: MessagesState) -> Literal["environment", END]:
```

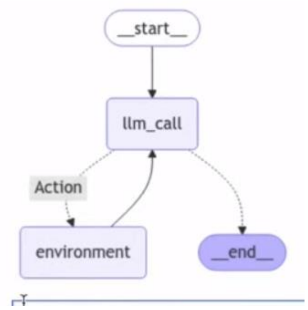


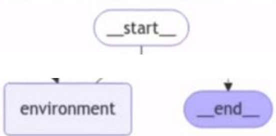
```python
# Conditional edge function to route to the tool node or end based upon whether the LLM made a tool call
def should_continue(state: MessagesState) -> Literal["environment", END]:
    """Decide if we should continue the loop or stop based upon whether the LLM made a tool call"""

    messages = state["messages"]
    last_message = messages[-1]
    # If the LLM makes a tool call, then perform an action
    if last_message.tool_calls:
        return "Action"
    # Otherwise, we stop (reply to the user)
    return END
```

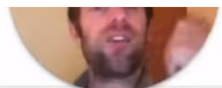
[44]:

```python
# Build workflow
agent_builder = StateGraph(MessagesState)

# Add nodes
agent_builder.add_node("llm_call", llm_call)
agent_builder.add_node("environment", tool_node)

# Add edges to connect nodes
agent_builder.add_edge(START, "llm_call")
agent_builder.add_conditional_edges(
    "llm_call",
    should_continue,
    {
        # Name returned by should_continue : Name of next node to visit
        "Action": "environment",
        END: END,
    },
)
agent_builder.add_edge("environment", "llm_call")

# Compile the agent
agent = agent_builder.compile()

# Show the agent
display(Image(agent.get_graph(xray=True).draw_mermaid_png()))
```





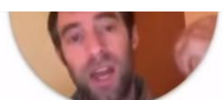




[45]:

```python
messages = [HumanMessage(content="Add 3 and 4. Then, take the output and multiple by 4.")]
messages = agent.invoke({"messages": messages})
for m in messages["messages"]:
    m.pretty_print()
```

```
================================ Human Message =================================

Add 3 and 4. Then, take the output and multiple by 4.
================================== Ai Message ==================================
```



```
================================ Human Message =================================

Add 3 and 4. Then, take the output and multiple by 4.
================================== Ai Message ==================================

[{'text': "I'll help you with that calculation. Let's break it down into steps:\n\n1. First, let's add 3 and 4:", 'type
d': 'toolu_019Lf9QGMkRogQEkK7k57hyB', 'input': {'a': 3, 'b': 4}, 'name': 'add', 'type': 'tool_use'}]
Tool Calls:
  add (toolu_019Lf9QGMkRogQEkK7k57hyB)
 Call ID: toolu_019Lf9QGMkRogQEkK7k57hyB
  Args:
    a: 3
    b: 4
================================= Tool Message =================================

7
================================== Ai Message ==================================

[{'text': "2. Now, let's multiply the result (7) by 4:", 'type': 'text'}, {'id': 'toolu_01UW8J5TMsv9qH11zTGSLzmW', 'inpu
'b': 4}, 'name': 'multiply', 'type': 'tool_use'}]
Tool Calls:
  multiply (toolu_01UW8J5TMsv9qH11zTGSLzmW)
 Call ID: toolu_01UW8J5TMsv9qH11zTGSLzmW
  Args:
    a: 7
    b: 4
================================= Tool Message =================================

28
================================== Ai Message ==================================

The final result is 28. Here's how we got there:
- 3 + 4 = 7
- 7 × 4 = 28
```

Pre-built

We also have a **pre-built method** for creating an agent as defined above (using the `create_react_agent` method):

https://langchain-ai.github.io/langgraph/how-tos/create-react-agent/

```
from langgraph.prebuilt import create_react_agent

# Pass in:
# (1) the augmented LLM with tools
# (2) the tools list (which is used to create the tool node)
pre_built_agent = create_react_agent(llm_with_tools, tools=tools)

# Show the agent
display(Image(pre_built_agent.get_graph().draw_mermaid_png()))
```

API Reference: create_react_agent