
LangGraph

Type Annotations

Dictionary📖

Normal Dictionary:

```
movie = {"name": "Avengers Endgame", "year": 2019}
```

- Allows for efficient data retrieval based on unique keys
- Flexible and easy to implement
- Leads to challenges in ensuring that the data is a particular structure, especially for larger projects
- Doesn't check if the data is the correct type or structure

Typed Dictionary⌨️

```python
from typing import TypedDict

class Movie(TypedDict):
    name : str
    year : int

movie = Movie(name="Avengers Endgame", year=2019)
```

- Type Safety - we defined explicitly what the data structures are, reducing runtime errors
- Enhanced Readability - Makes debugging easier and makes code more understandable.

Union 🤝

```python
from typing import Union

def square(x: Union[int, float]) -> float:
    return x * x

x = 5       # ✅ this is fine because it is an integer
x = 1.234   # ✅ this is also fine because it is a float
x = "I am a string!"    # ❌ this will fail because it is a string
```

- Union lets you say that a value can be more than one type
- Flexible and easy to code
- Type Safety as it can provide hints to help catch incorrect usage

Optional 🤔

```python
from typing import Optional

def nice_messasge(name: Optional[str]) -> None:
    if name is None:
        print("Hey random person!")
    else:
        print(f"Hi there, {name}!")
```

- In this case “name” can be either String or None!
- It cannot be anything else

Any🎲

```python
from typing import Any

def print_value(x: Any):
    print(x)

print_value("I pretend to be Batman in the shower sometimes")
```

- Anything and everything is allowed!

Lambda Function ⏳

```
square = lambda x: x * x
square(10)
```

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x * x, nums))
```

- Lambda is just a shortcut to writing small functions!

Elements

State 🧮

➔ The **State** is a shared data structure that holds the current information or context of the entire application.

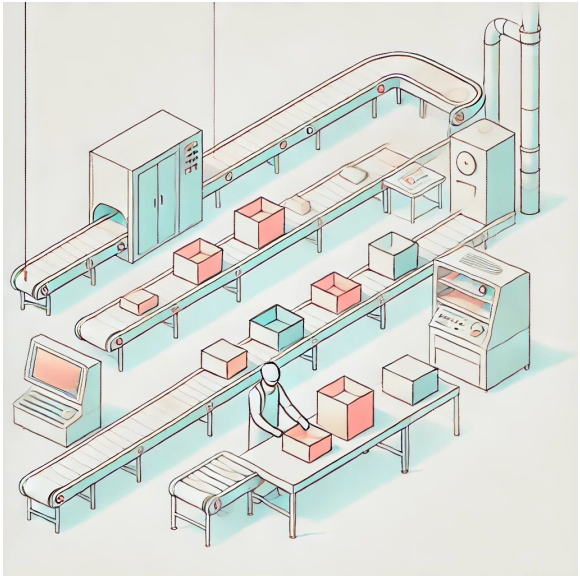
➔ In simple terms, it is like the application's memory, keeping track of the variables and data that nodes can access and modify as they execute.

Analogy:

- **Whiteboard in a Meeting Room:** Participants (nodes) write and read information on the whiteboard (state) to stay updated and coordinate actions.

Nodes 📌

- ➔ **Nodes** are individual functions or operations that perform specific tasks within the graph.
- ➔ Each node receives input (often the current state), processes it, and produces an output or an updated state.

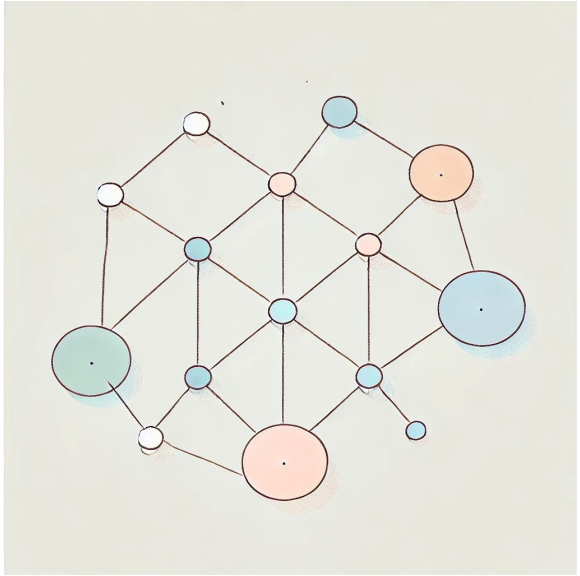


Analogy:

- **Assembly Line Stations:** Each station does one job—attach a part, paint it, inspect quality, and so on.

Graph 🍡

- ➔ A **Graph** in LangGraph is the overarching structure that maps out how different tasks (nodes) are connected and executed.
- ➔ It visually represents the workflow, showing the sequence and conditional paths between various operations.

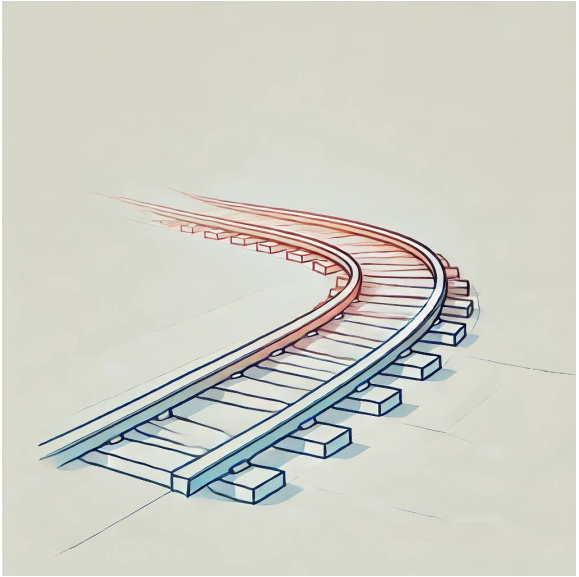


Analogy:

- **Road Map:** A road map displaying the different routes connecting cities, with intersections offering choices on which path to take next.

Edges 🔥

- ➔ **Edges** are the connections between nodes that determine the flow of execution.
- ➔ They tell us which node should be executed next after the current one completes its task.

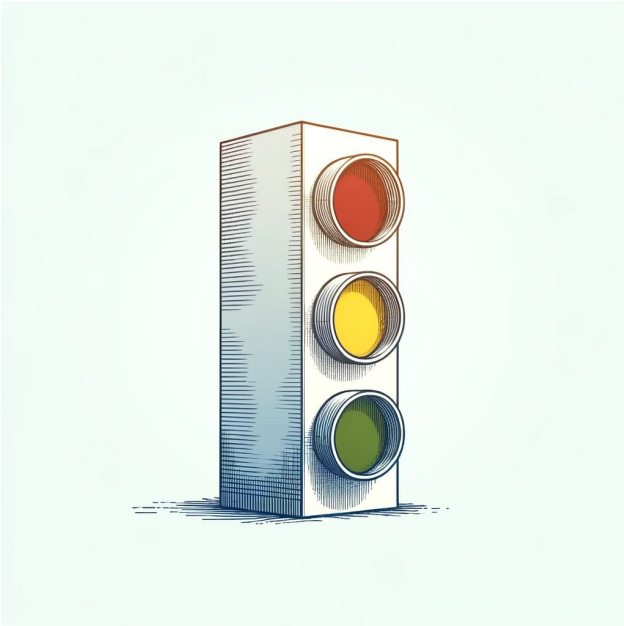


Analogy:

- **Train Tracks:** Each track (edge) connects the stations (nodes) together in a specific direction.

Conditional Edges 🎏

➔ **Conditional Edges** are specialized connections that decide the next node to execute based on specific conditions or logic applied to the current state.



Analogy:

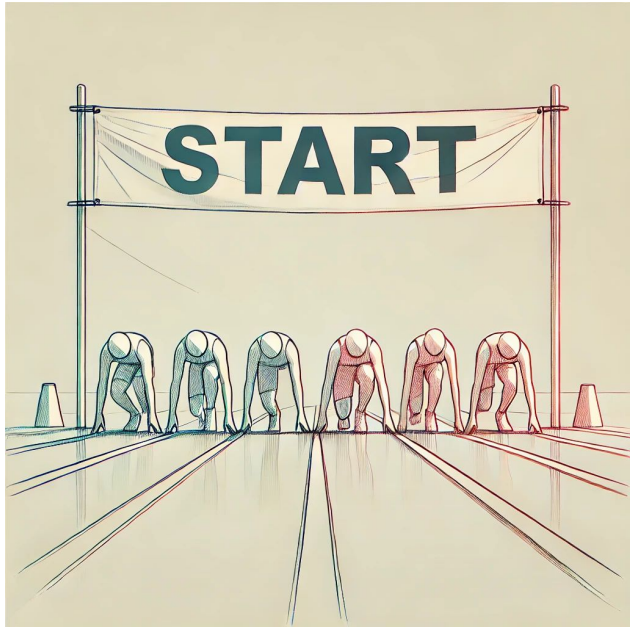
- **Traffic Lights:** Green means go one way, red means stop, yellow means slow down. The condition (light color) decides the next step.

START 🚀

- ➔ The **START** node is a virtual entry point in LangGraph, marking where the workflow begins.
- ➔ It doesn't perform any operations itself but serves as the designated starting position for the graph's execution.

Analogy:

- **Race Starting Line:** The place where a race officially begins.



END 🚩

- ➔ The **END** node signifies the conclusion of the workflow in LangGraph.
- ➔ Upon reaching this node, the graph's execution stops, indicating that all intended processes have been completed.

Analogy:

- **Finish Line in a Race:** The race is over when you cross it.

Tools🛠️

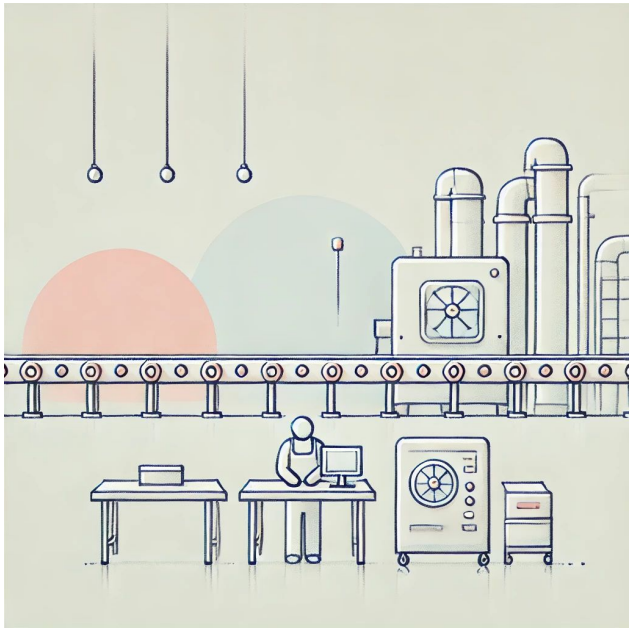
- ➔ **Tools** are specialized functions or utilities that nodes can utilize to perform specific tasks such as fetching data from an API.
- ➔ They enhance the capabilities of nodes by providing additional functionalities.
- ➔ Nodes are part of the graph structure, while tools are functionalities used within nodes

Analogy:

- **Tools in a Toolbox:** A hammer for nails, a screwdriver for screws, each tool has a distinct purpose.

ToolNode🔎

- ➔ A **ToolNode** is just a special kind of node whose main job is to run a tool.
- ➔ It connects the tool's output back into the State, so other nodes can use that information.



Analogy:

- **Operator Using a Machine:** The operator (ToolNode) controls the machine (Tool), then takes the results back to the assembly line.

StateGraph🏤

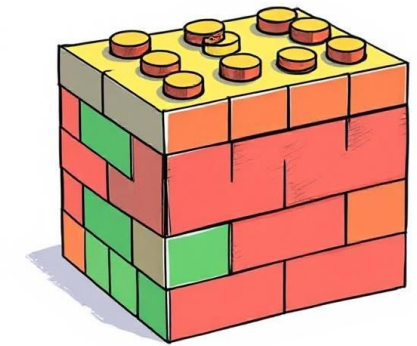
- ➔ A **StateGraph** is a class in LangGraph used to build and compile the graph structure.
- ➔ It manages the nodes, edges, and the overall state, ensuring that the workflow operates in a unified way and that data flows correctly between components.

Analogy:

- **Blueprint of a Building:** Just as a blueprint outlines the design and connections within a building, a StateGraph defines the structure and flow of the workflow.

Runnable⚡

- ➔ A Runnable in LangGraph is a standardized, executable component that performs a specific task within an AI workflow.
- ➔ It serves as a fundamental building block, allowing for us to create modular systems.



Analogy:

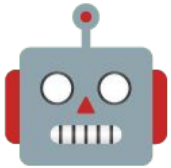
- **LEGO Brick:** Just as LEGO bricks can be snapped together to build complex structures, Runnables can be combined to create sophisticated AI workflows.

Messages📬

Human Message
Represents input from a user.

System Message
Used to provide instructions or context to the model

Function Message
Represents the result of a function call



AI Message
Represents responses generated by AI models

Tool Message
Similar to Function Message, but specific to tool usage

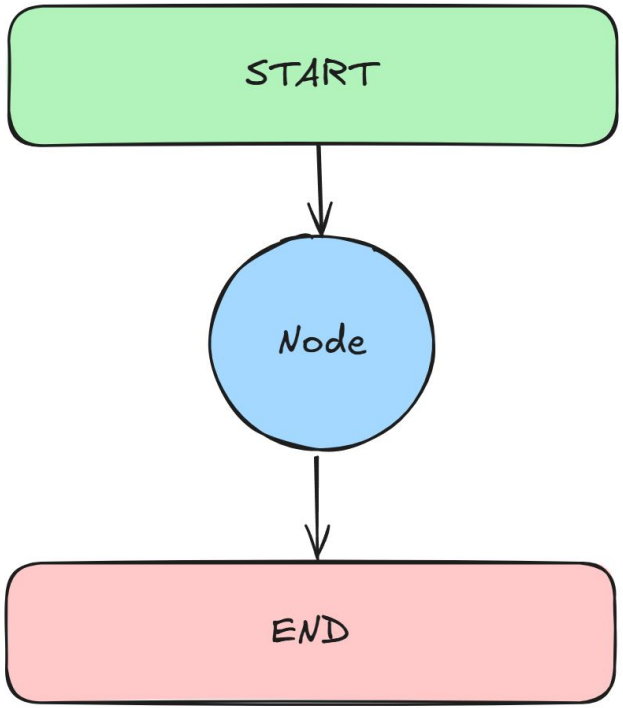
Graph I

not yet… but soon!

Hello World Graph🌍

Objectives ✅:

1. Understand and define the AgentState structure
2. Create simple node functions to process and update state
3. Set up a basic LangGraph structure
4. Compile and invoke a LangGraph graph
5. Understand how data flows through a single-node in LangGraph


START
Node
END

Exercise for Graph I 🏋️

🏗️Your task:

Create a **Personalized Compliment Agent** using LangGraph!

Input: {"name": "Bob"}

Output: "Bob, you're doing an amazing job learning LangGraph!"

Hint: You have to concatenate the state, not replace it!

Graph II

Multiple Inputs Graph🔢

Objectives ✅:

1. Define a more complex AgentState
2. Create a processing node that performs operations on **list data.**
3. Set up a LangGraph that processes and outputs computed results.
4. Invoke the graph with structured inputs and retrieve outputs.

Main Goal: Learn how to handle multiple inputs

Exercise for Graph II 🏆

🏗️Your task:

Create a **Graph** where you pass in a single list of integers along with a name and an operation. If the operation is a "+", you **add** the elements and if it is a "*", you **multiply** the elements, **all within the same node.**

Input: {"name": "Jack Sparrow","values": [1,2,3,4] , "operation": "*"}

Output: "Hi Jack Sparrow, your answer is: 24"

Hint: You need an if-statement in your node!

Graph III

Sequential Graph♾️

Objectives:

1. Create **multiple Nodes** that sequentially process and update different parts of the state.
2. Connect Nodes together in a graph
3. Invoke the Graph and see how the **state is transformed** step-by-step.

Main Goal: Create and handle multiple Nodes

Exercise for Graph III 💪

Your task:

1. Accept a user's name, age, and a list of their skills.
2. Pass the state through **three nodes** that:
    - **First node:** Personalizes the name field with a greeting.
    - **Second node:** Describes the user's age.
    - **Third node:** Lists the user's skills in a formatted string.
3. The final output in the result field should be a **combined message** in this format:

Output: "Linda, welcome to the system! You are 31 years old! You have skills in: Python, Machine Learning, and LangGraph"

Hint: You will need to use the the add_edge method twice

Graph IV

Conditional Graph🚧

Objectives:

1. Implement conditional logic to route the flow of data to different nodes
2. Use **START** and **END** nodes to manage entry and exit points explicitly.
3. Design multiple nodes to perform different operations (addition, subtraction).
4. Create a router node to handle decision-making and control graph flow.

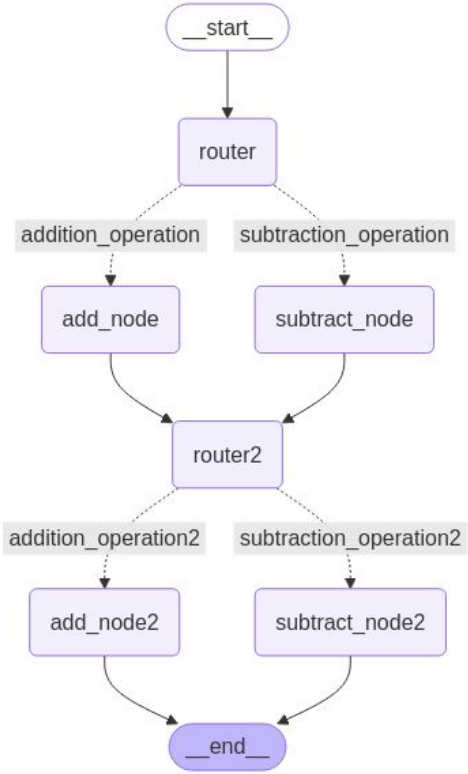
Main Goal: How to use "add_conditional_edges()"

Exercise for Graph IV 🛹

Your task:

Make the graph on the right! You will need to make use of 2 conditional edges!

Input: initial_state = AgentState(number1 = 10, operation="-", number2 = 5, number3 = 7, number4=2, operation2="+", finalNumber= 0, finalNumber2 = 0)



Graph V

Looping Graph🌌

Objectives:

1. Implement looping logic to route the flow of data back to the nodes
2. Create a single conditional edge to handle decision-making and control graph flow.

Main Goal: Coding up Looping Logic

Exercise for Graph V 🧰

Your task:

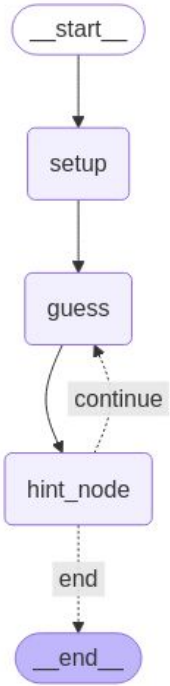
Make the graph on the right! You need to implement an Automatic Higher or Lower Game.

Set the bounds to between 1 to 20. The Graph has to keep guessing (max number of guesses is 7) where if the guess is correct, then it stops, but if not we keep looping until we hit the max limit of 7.

Each time a number is guessed, the hint node should say higher or lower and the graph should account for this information and guess the next guess accordingly.

Input: {"player_name": "Student", "guesses": [], "attempts": 0, "lower_bound": 1, "upper_bound": 20}

Hint: It will need to adjust its bounds after every guess based on the hint provided by the hint node.



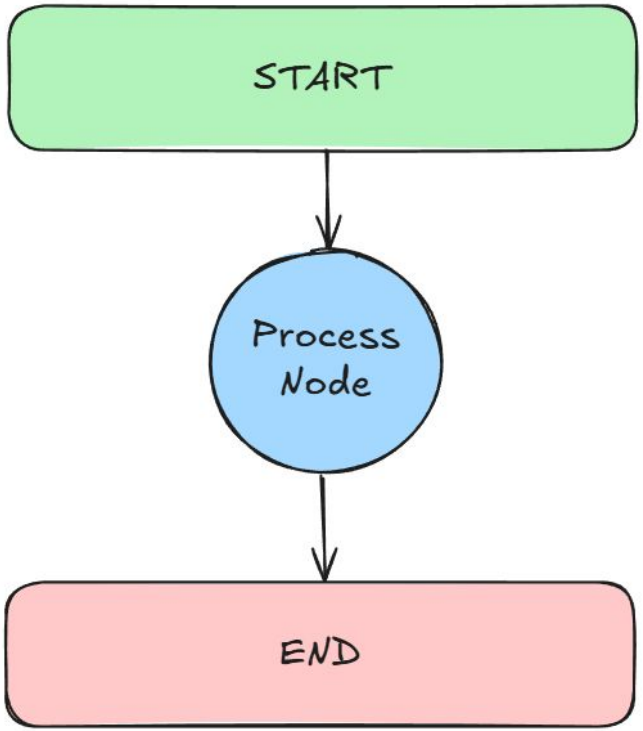
Agent I

Simple Bot🤖

Objectives:

1. Define state structure with a list of HumanMessage objects.
2. Initialize a GPT-4o model using LangChain's ChatOpenAI
3. Sending and handling different types of messages
4. Building and compiling the graph of the Agent

Main Goal: How to integrate LLMs in our Graphs


START
Process Node
END

Agent II

Chatbot🧠

Objectives:

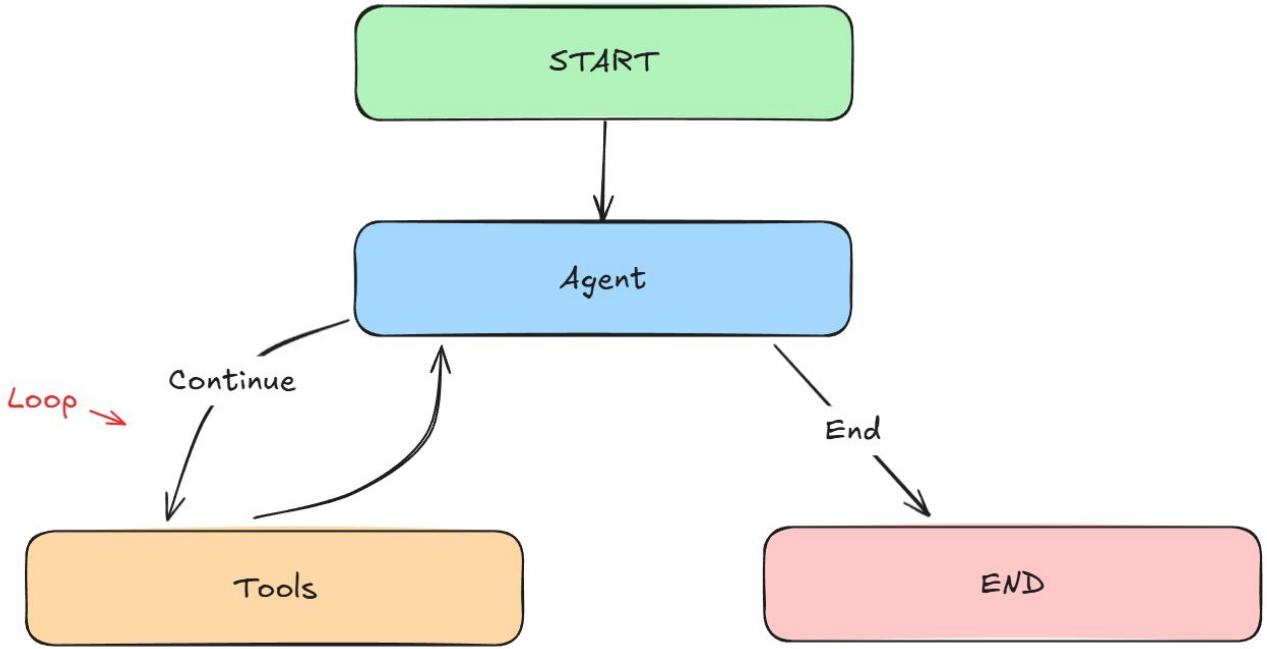
1. Use different message types - HumanMessage and AIMessage
2. Maintain a full conversation history using both message types
3. Use GPT-4o model using LangChain's ChatOpenAI
4. Create a sophisticated conversation loop

Main Goal: Create a form of memory for our Agent

Agent III

ReAct Agent

Reasoning and Acting Agent


START
Agent
Continue
Loop
End
Tools
END

ReAct Agent 📱

Objectives:

1. Learn how to create Tools in LangGraph
2. How to create a ReAct Graph
3. Work with different types of Messages such as ToolMessages
4. Test out robustness of our graph

Main Goal: Create a robust ReAct Agent!

Agent IV

DRAFTER

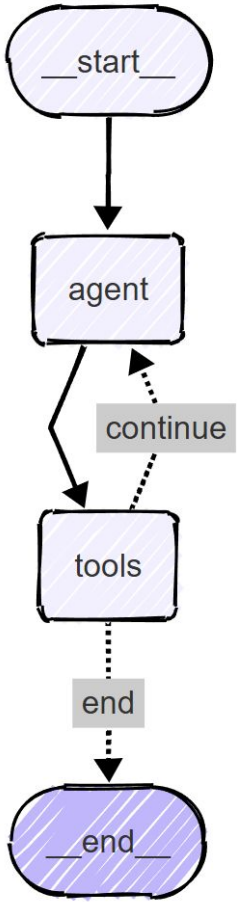
I put it in red colour to give it a more dramatic effect!

Boss's Orders 👔

Task:

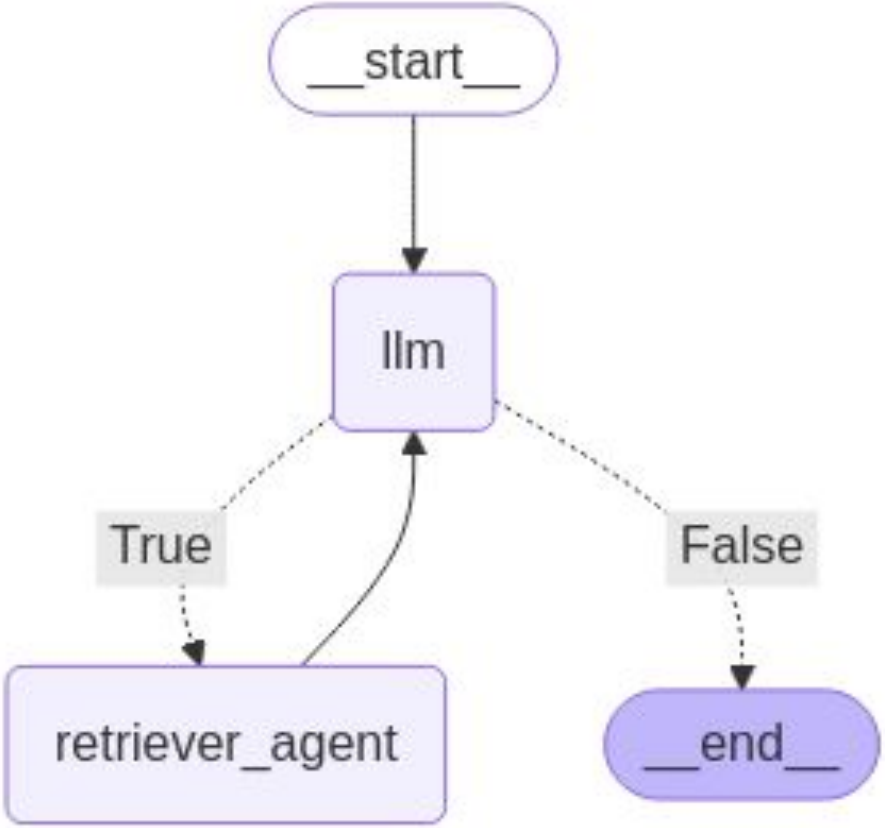
Our company is not working efficiently! We spend way too much time drafting documents and this needs to be fixed! ⏱️

For the company, you need to create an AI Agentic System that can speed up drafting documents, emails, etc. The AI Agentic System should have Human-AI Collaboration meaning the Human should be able to able to provide continuous feedback and the AI Agent should stop when the Human is happy with the draft. The system should also be fast and be able to save the drafts.


__start__
agent
continue
tools
end
__end__

Agent V

RAG


__start__
llm
True
False
retriever_agent
__end__