

Complete LangGraph Crash Course



Advanced

Intermediate

Basic

Course Overview

1. Levels of Autonomy in LLM applications (Code -> LLM Call -> Chain -> Router -> Agent)
2. Understanding Agents & Tools
3. Building Agents & Tools from Scratch
4. Building Agents From pre-defined LangChain classes

Course Overview

5. Graph Structure

- Direct Acyclic Graph (DAG) vs Cyclic Graph

6. What is LangGraph?

7. Why LangGraph is required?

8. Creating LangGraph from scratch

9. Creating a LangGraph using in-built classes (Reflection, Reflexion agents, etc.)

10. Key concepts and terms in LangGraph

- Graph, state, nodes, edges, visualisation, checkpoints, breakpoints, configuration

Course Overview

11. Creating a Chatbot with LangGraph

12. Common Agentic Patterns

- Human-in-the-loop
- ReAct Agent, and many more.

13. Multi-agent systems using LangGraph

14. RAGs with LangGraph: CRAG vs ARAG vs self-RAG

15. Persistence

Course Overview

16. LangGraph ecosystem

- LangGraph Studio
- LangGraph Cloud API, etc

17. Agents in Production

Pre-requisites

1. You need to have Python 3.8 or higher installed
2. LangChain (as LangGraph builds on top of LangChain)

Levels of Autonomy in LLM applications

1. Code

Code has zero autonomy and is 100% deterministic

We all know that everything is hard-coded and it is not even really a cognitive architecture.

Disadvantage:

The problem? You'd need to write rules for every possible scenario - making it impossible to handle real-world complexity.

Levels of Autonomy in LLM applications

2. LLM call

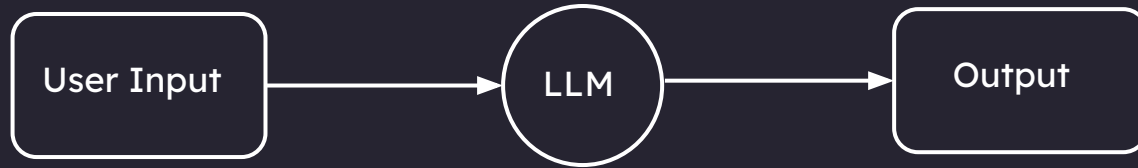
A single LLM call means your app basically does one main thing - you give it an input, it processes it, and gives you back an output.

Think of chatbots that just take your message and respond, or apps that translate text.

This was a huge leap from hard-coded rules, even though it's still pretty simple and is only in the 2nd stage of autonomy

Levels of Autonomy in LLM applications

2. LLM call



Example User Input: You are an expert LinkedIn post writer. Write me a post on "AI Agents Taking over Content Creation"

Levels of Autonomy in LLM applications

2. LLM call

A single LLM call means your app basically does one main thing - you give it an input, it processes it, and gives you back an output.

Think of chatbots that just take your message and respond, or apps that translate text.

This was a huge leap from hard-coded rules, even though it's still pretty simple and is only in the 2nd stage of autonomy

Disadvantage:

Trying to get everything done in one shot often leads to confused or mixed-up responses - just like how a single person can't be an expert at everything.

Levels of Autonomy in LLM applications

3. Chains

Think of chains like having multiple specialists instead of one generalist. Instead of asking one AI to do everything, we break it down into steps where each AI is really good at one thing.

Imagine a customer service chatbot: The first AI reads your complaint and figures out exactly what product you're talking about

The second AI finds the right solution from the company's help docs, and the third AI turns that solution into a friendly response.

Each step is simple, but together they create a much smarter system than a single LLM call could.

Levels of Autonomy in LLM applications

3. Chains (contd.)

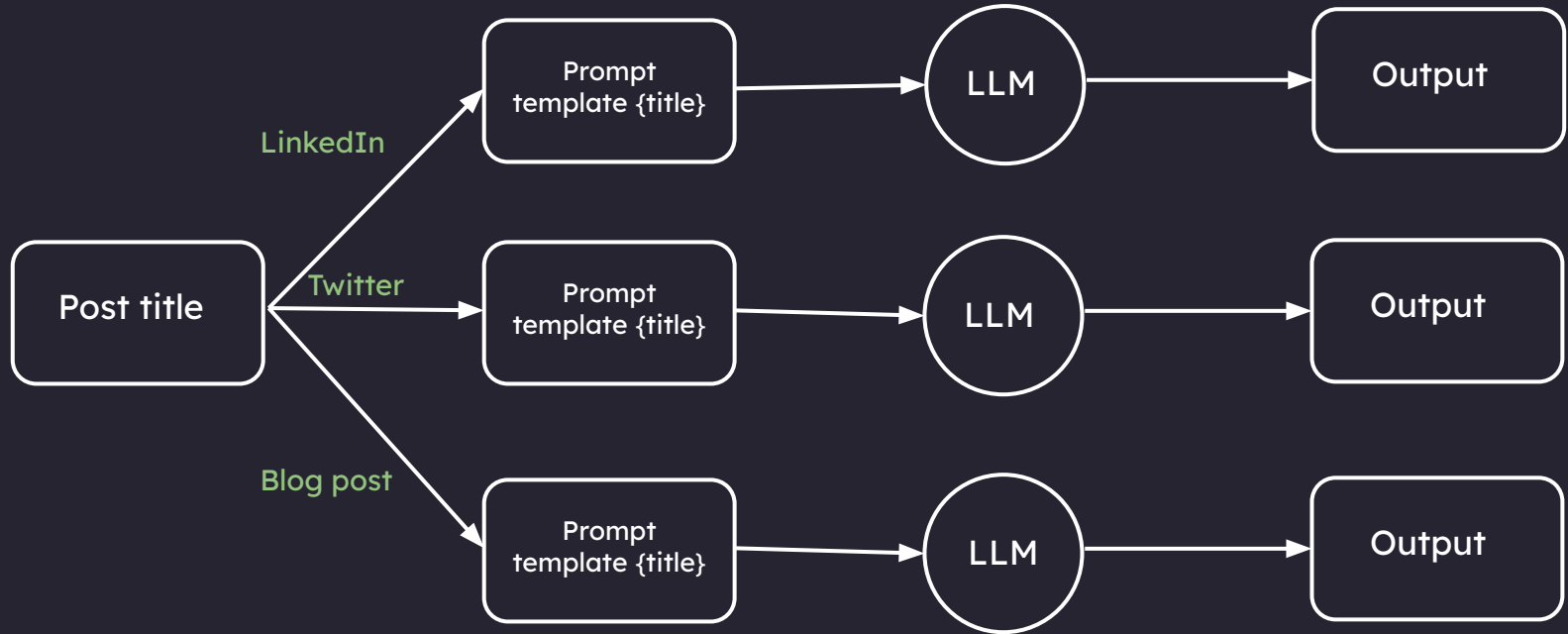
This is where we first started seeing AI apps that could handle more complex tasks - not just by being smarter, but by breaking big problems into smaller, manageable pieces.

Disadvantage:

The downside? These fixed sequences are like a rigid assembly line - they always follow the same steps defined by the human.

Levels of Autonomy in LLM applications

3. Chains (contd.)



Example User Input: "AI Agents taking over Content Creation"

Levels of Autonomy in LLM applications

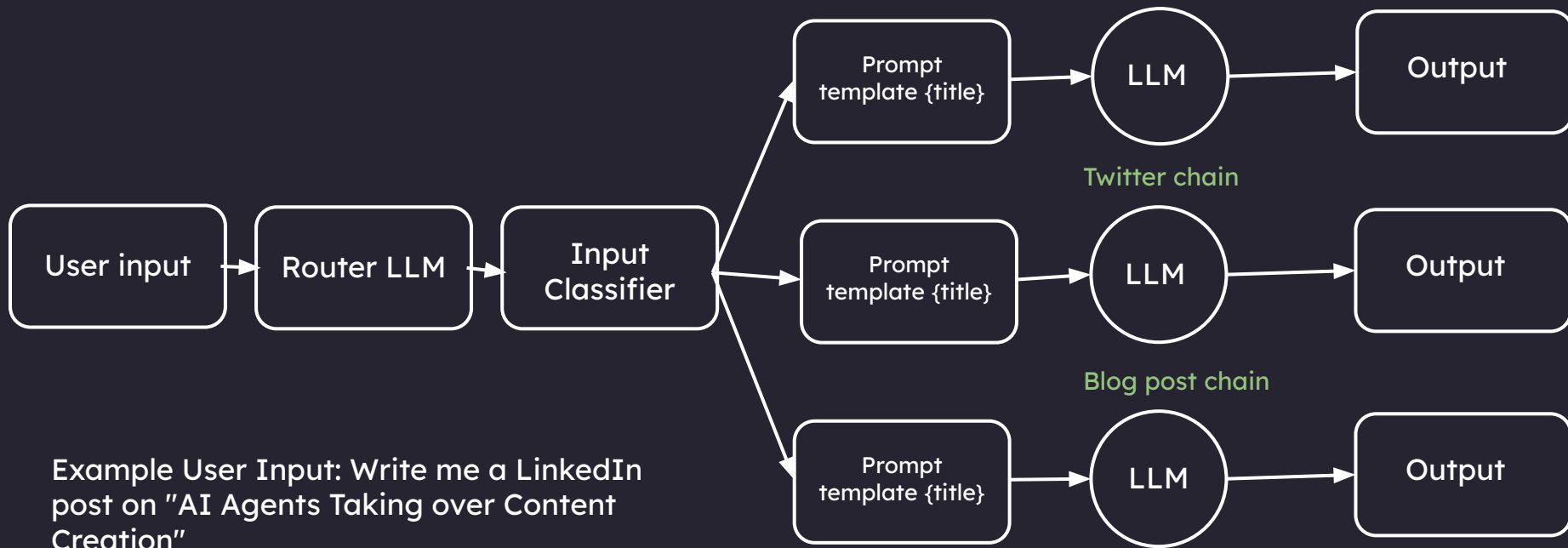
4. Router

Now this is where it gets interesting - routers are like smart traffic cops for your AI. Instead of having a fixed path like in chains, the AI itself decides what steps to take next.

Imagine a personal assistant bot: when you ask it something, it first figures out if you need help with scheduling, research, or calculations, then routes your request to the right tool or chain for the job.

Levels of Autonomy in LLM applications

4. Router



Levels of Autonomy in LLM applications

4. Router

Now this is where it gets interesting - routers are like smart traffic cops for your AI. Instead of having a fixed path like in chains, the AI itself decides what steps to take next.

Imagine a personal assistant bot: when you ask it something, it first figures out if you need help with scheduling, research, or calculations, then routes your request to the right tool or chain for the job.

Disadvantage:

While it can choose different paths, it still can't remember previous conversations or learn from mistakes.

Levels of Autonomy in LLM applications

5. State Machine (Agent)

This is combining the previous level (router) but with loops.

Agent ~= control flow controlled by an LLM

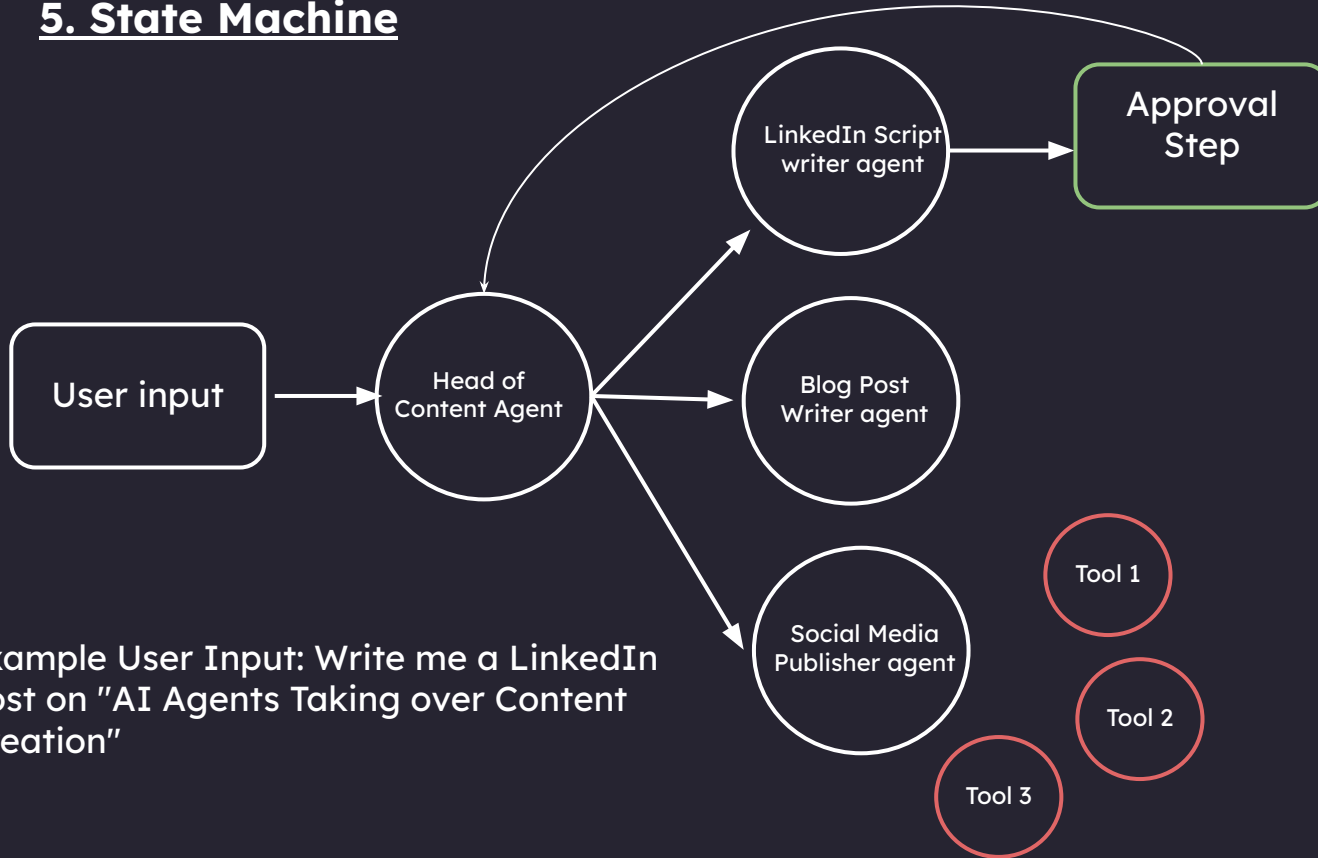
This involves features like:

1. Ability to have human-in-loop, ask for approval before moving on
2. Multi-agent systems
3. Advanced memory management
4. Go back in history and explore better alternate paths
5. Adaptive Learning

And many more, and THIS is where LangGraph comes into the picture

Levels of Autonomy in LLM applications

5. State Machine















Levels of autonomy in LLM applications



code



LLMs

		Decide Output of Step	Decide Which Steps to Take	Decide What Steps are Available to Take
HUMAN-DRIVEN	1	Code		
	2	LLM Call	 <i>one step only</i>	
	3	Chain	 <i>multiple steps</i>	
	4	Router	 <i>no cycles</i>	
<hr/>				
AGENT-EXECUTED	5	State Machine	 <i>cycles</i>	
	6	Autonomous		



LangChain

Levels of Autonomy in LLM applications

Chain/Router vs Agent

A Chain or even a router is one directional. Hence, it is not an agent

Whereas in a state machine, we can go back in the chain, have cycles and the flow is controlled by the LLM, hence it is called an Agent

Understanding AI Agents

AI Agents & Tools

Think of Agents as the "problem-solvers" of the AI world. Agents are capable of thinking on their own.

In other words, it's AI that can make autonomous decisions.

In the case of Chains and Router, they follow our specific instruction.

But with agents, they actually take it a step further. They can decide for themselves what steps to take on their own.

AI Agents & Tools

What are tools then?

Tools are specific functions that Agents can use to complete tasks

Just like a chef's kitchen tools (knife for cutting, oven for baking, blender for mixing), tools are the special abilities we give to AI - like giving it a calculator tool, or a search engine tool, or a calendar tool

AI Agents & Tools (Re-Act Agent Pattern)

This is one of the best known patterns in AI today to build agents. It stands for Reasoning + Acting

This is basically a concept that mimics how human beings think.

ReACT pattern

Think: LLM first thinks about the user prompt/problem

Action: LLM decides if it can answer by itself or if it should use a tool

Action Input: LLM provides the input argument for the tool

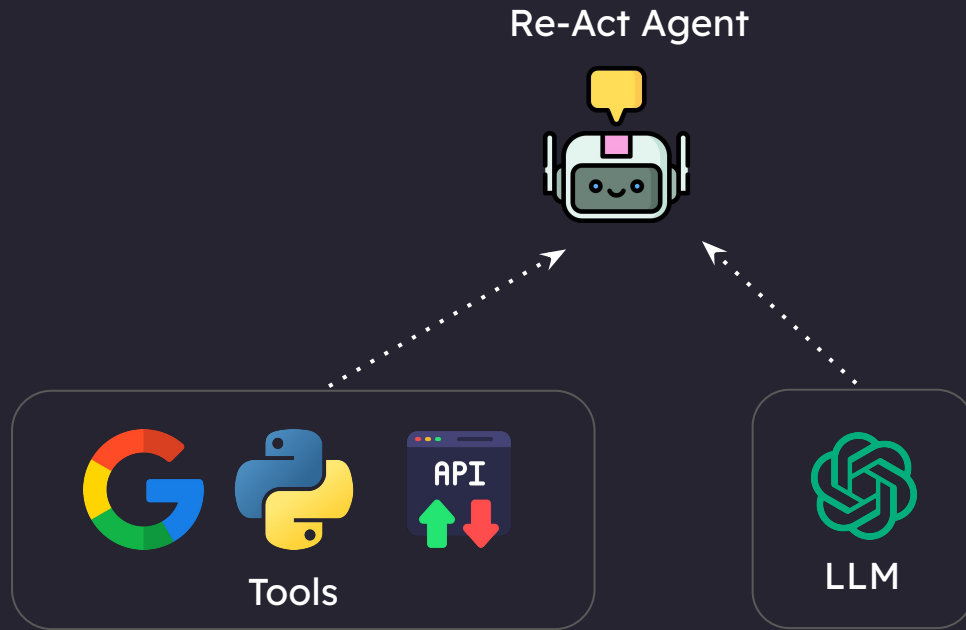
Here, langChain executes the tool and returns the output to the LLM

Observe: LLM observe the result of the tool

⋮

Final Answer: "This is your final answer"

ReACT pattern

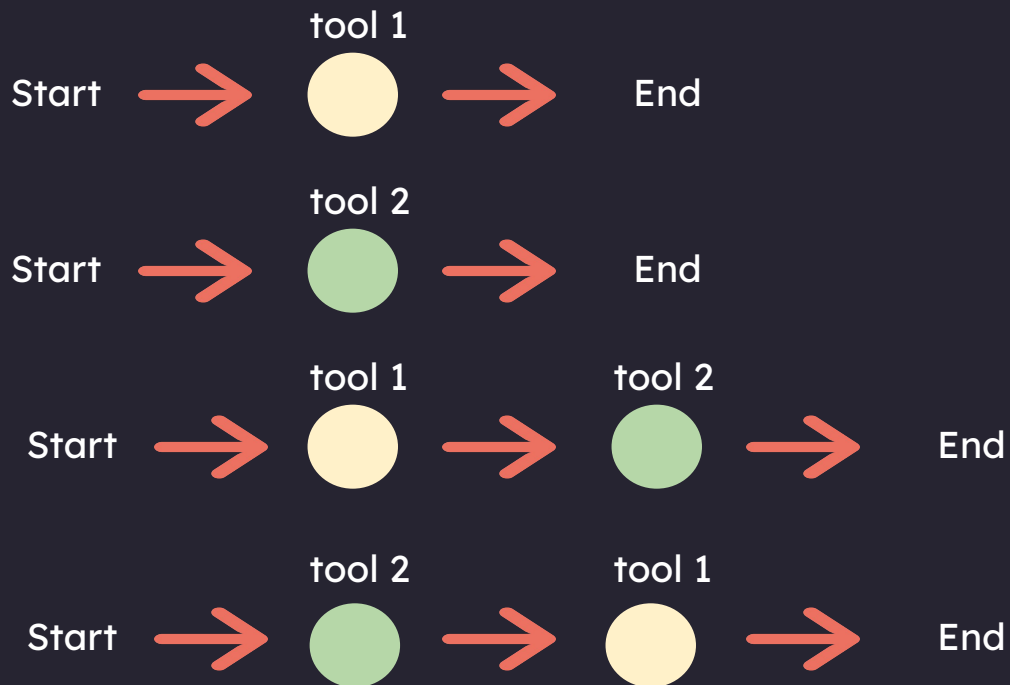


Let's Jump Into The Code

1. Build a Re-act Agent Using LangChain
2. What are its drawbacks and where does LangGraph come into the picture?

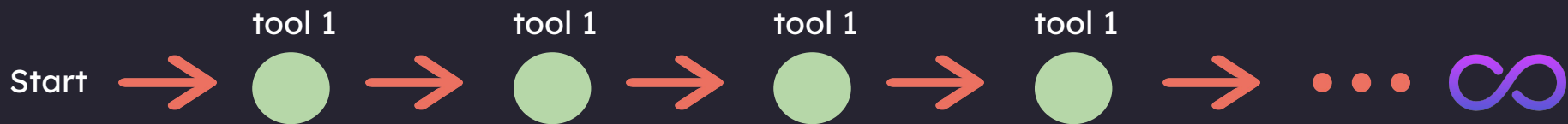
ReAct Agents

ReAct Agents Are Flexible. i.e., Any state is possible



ReAct Agents

But high flexibility can also mean less reliability



Infinite Loop causes:

1. We did not define the tools correctly
2. The LLM is not capable enough
3. The prompting doesn't define a clear end condition

Best Of Both Worlds

Chain

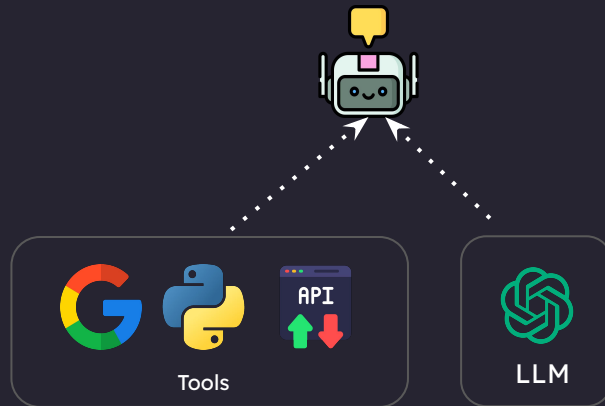


Not Flexible
More Reliable



Flexible &
Reliable

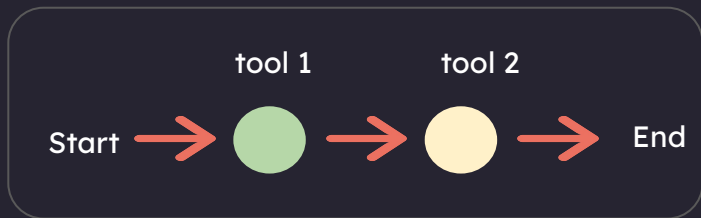
React Agent



Flexible
Less Reliable

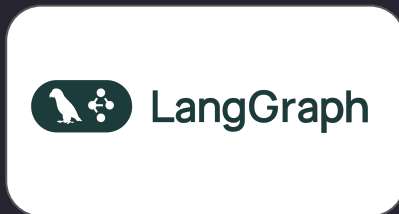
Best Of Both Worlds

Chain



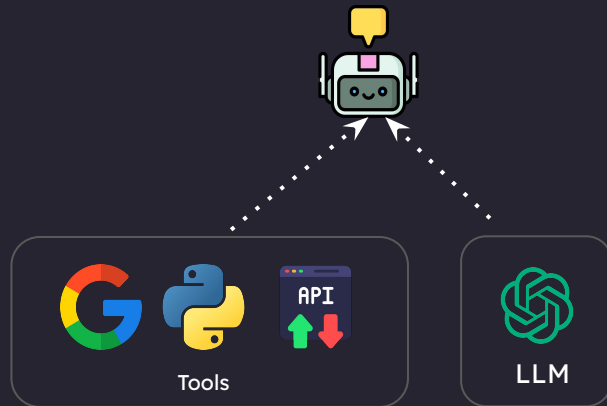
Not Flexible
More Reliable

LangGraph



Flexible &
Reliable

React Agent



Flexible
Less Reliable

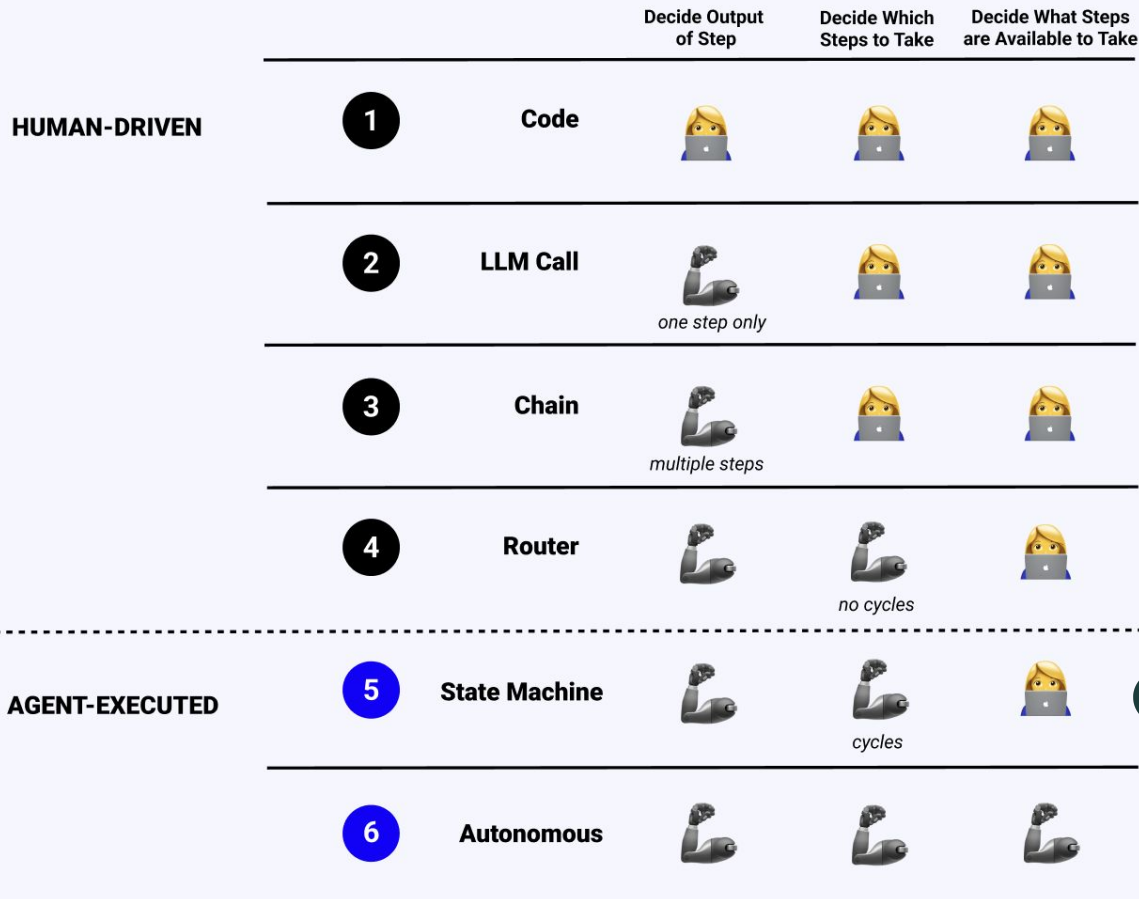
Levels of autonomy in LLM applications



code



LLMs



LangGraph

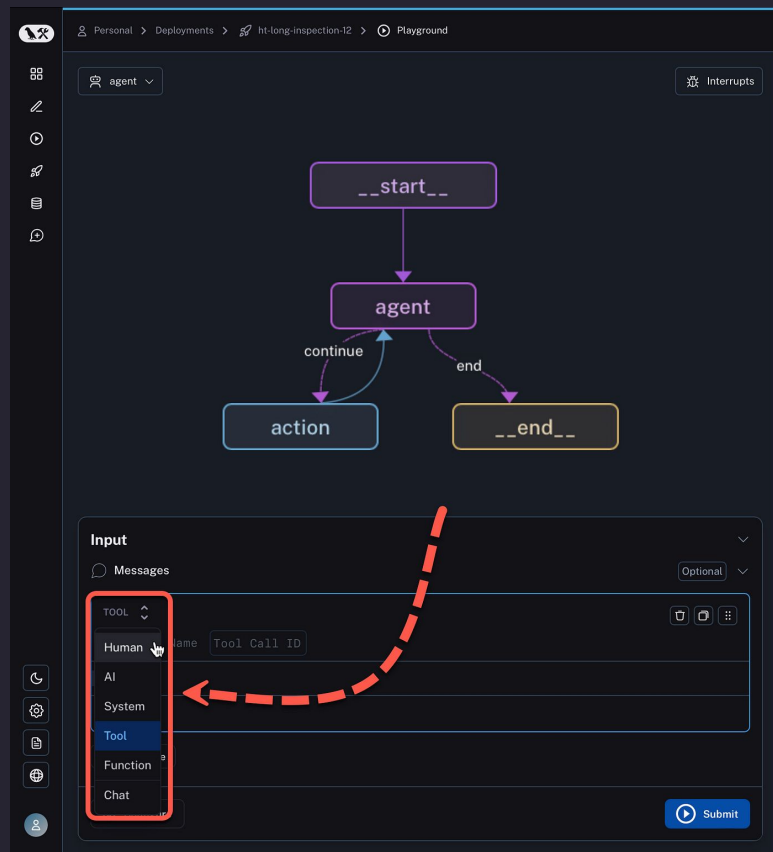


LangChain

What is LangGraph?

A framework for building controllable, persistent agent workflows with built-in support for human interaction, streaming, and state management.

It uses the Graph Data Structure to achieve this



Key Features Of LangGraph

1. Looping and Branching Capabilities:

Supports conditional statements and loop structures, allowing dynamic execution paths based on state.

2. State Persistence:

Automatically saves and manages state, supporting pause and resume for long-running conversations.

3. Human-Machine Interaction Support:

Allows inserting human review during execution, supporting state editing and modification with flexible interaction control mechanisms.

Key Features Of LangGraph

4. Streaming Processing:

Supports streaming output and real-time feedback on execution status to enhance user experience.

5. Seamless Integration with LangChain:

Reuses existing LangChain components, supports LCEL expressions, and offers rich tool and model support.

Why use the Graph Data Structure?

Why Graphs?

Tree of Thoughts: Deliberate Problem Solving with Large Language Models



Figure 1: Schematic illustrating various approaches to problem solving with LLMs. Each rectangle box represents a thought, which is a coherent language sequence that serves as an intermediate step toward problem solving. See concrete examples of how thoughts are generated, evaluated, and selected in Figures 2, 4, 6.

Instruction Tuning for Large Language Models: A Survey

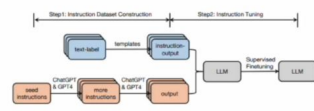


Figure 1: General pipeline of instruction tuning.

from the internet. The creation of these datasets typically involves machine learning techniques, relying solely on manual gathering and verification, resulting in generally smaller datasets. Below are some widely-used human-crafted datasets.

"Inputs", "answer_choices", and "targets" "Inputs" is a sequence of text that describes the task in natural language (e.g., "If he like Mary's cat", "Answer choices" is a list of text using that are applicable responses to the above task in a "Correct" "Yes"

SELF-REFINE: Iterative Refinement with Self-Feedback

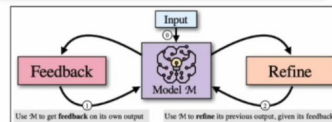


Figure 1: Given an input (1), SELF-REFINE starts by generating an output and passing it back to the same model M to get feedback (2). The feedback is passed back to M , which refines the previously generated output (3). Steps (1) and (2) iterate until a stopping condition is met. SELF-REFINE is instantiated with a language model such as GPT-3.5 and does not involve human assistance.

SELF-RAG: LEARNING TO RETRIEVE, GENERATE, AND CRITIQUE THROUGH SELF-REFLECTION

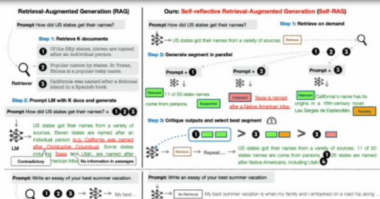


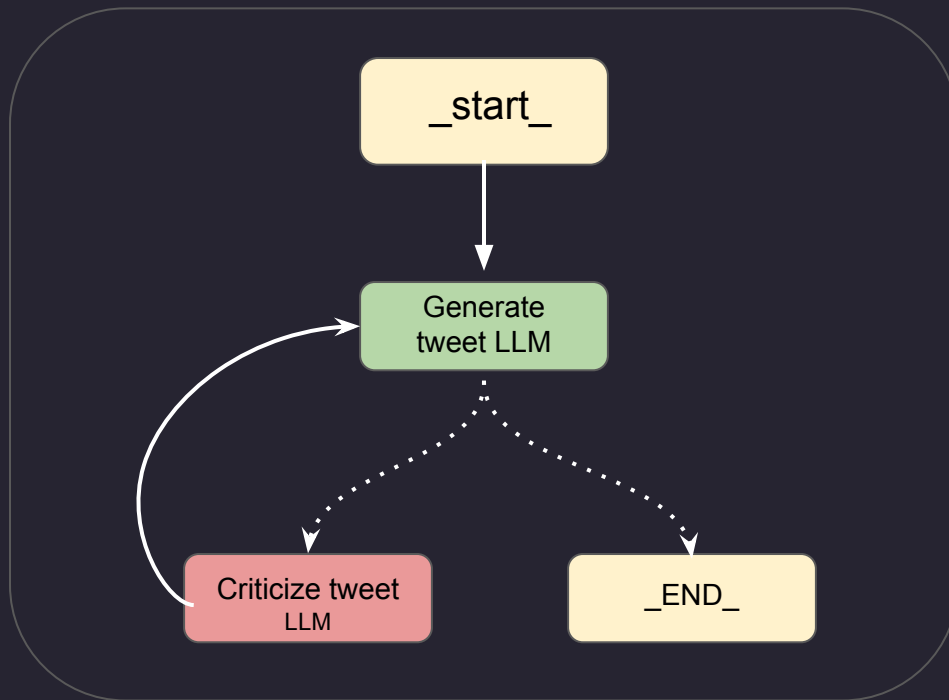
Figure 1: Overview of SELF-RAG. SELF-RAG learns to retrieve, critique, and generate text passages to enhance overall generation quality, factuality, and verifiability.

Core Components of LangGraph

Core Components of LangGraph

1. Nodes
2. Edges
3. Conditional Edges
4. State

Example: Reflection Agent pattern



Reflection Agents in LangGraph

Reflection Agents in LangGraph

1. What is a Reflection Agent System?
2. Three types of Reflection Agent Systems
3. Setup & Installations
4. Implement a reflection Agent System

Reflection Agent pattern in LangGraph

But what does the English word "reflection" mean?

Like how you're looking at your reflection in the mirror, reflection means looking at yourself or your actions

For example:

- After giving a presentation, thinking about how it went
- After writing an email, reading it again to check if it's clear
- After making a decision, considering if it was the right choice

Reflection Agent pattern in LangGraph

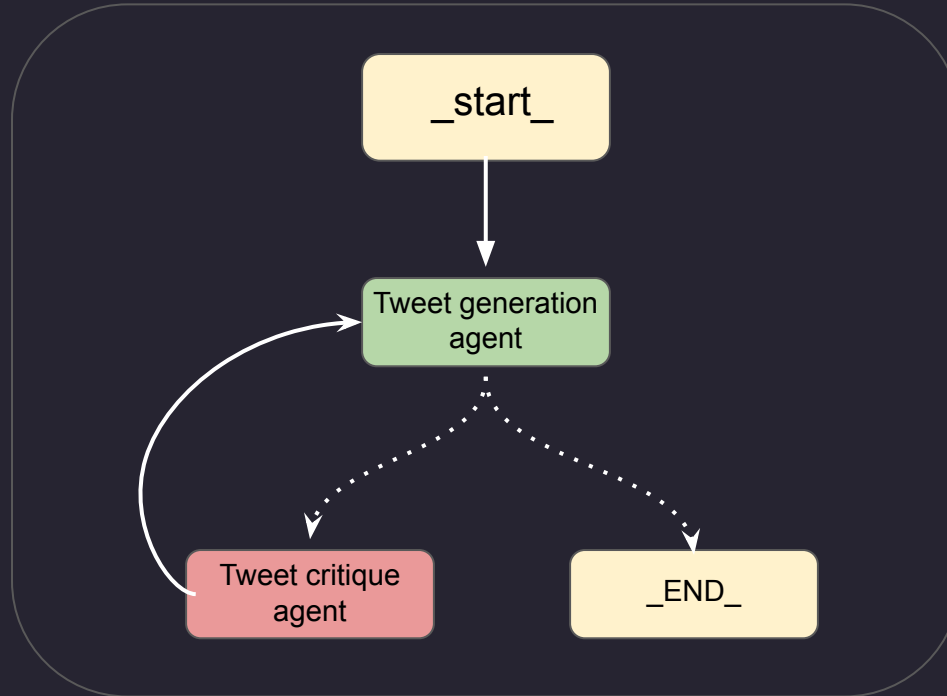
So what is a reflection-agent pattern?

A reflection agent pattern is an AI system pattern that can look at its own outputs and think about them/make it better - just like how we look at ourselves in a mirror and self-reflect, make ourselves better

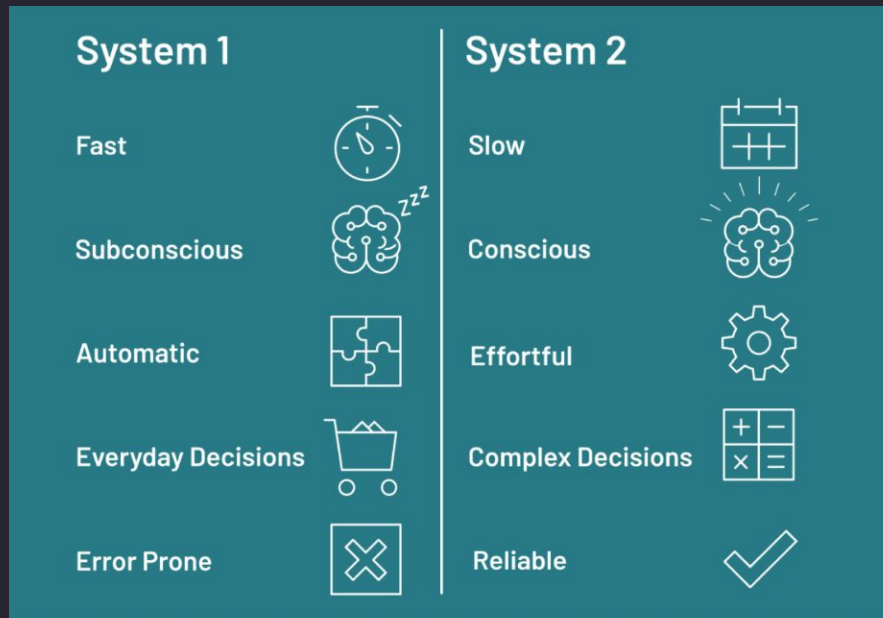
A basic reflection agent system typically consists of:

1. A generator agent
2. A reflector agent

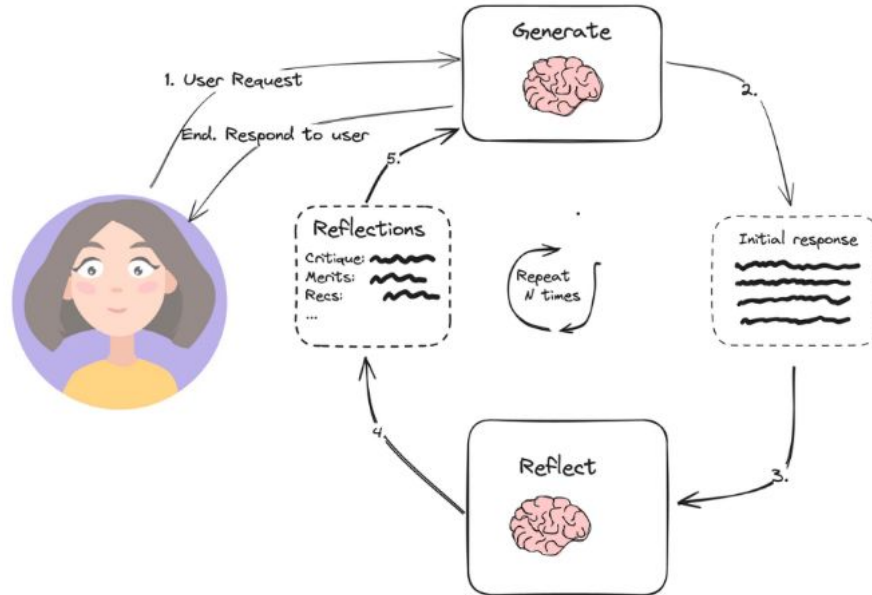
Example: Basic Reflection Agent pattern



Reflection Agent pattern in LangGraph



Basic Reflection



Simple Reflection Loop

Types of Reflection Agents in LangGraph

There are 3 types:

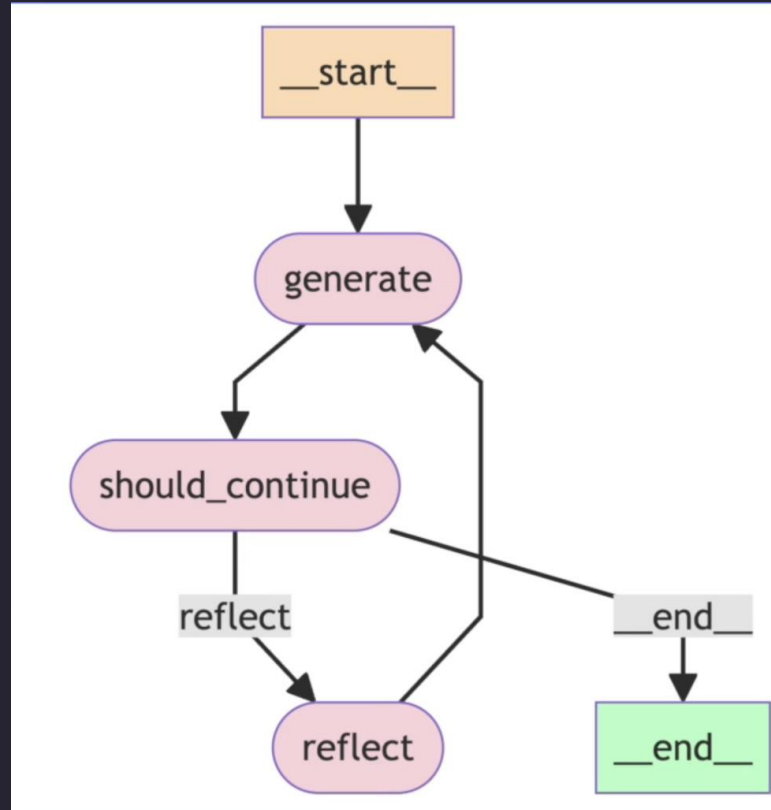
1. Basic Reflection Agents
2. Reflexion Agents
3. Language Agent Tree Search (LATS)

Let's Implement a Basic Reflection Agent!

Let's Implement a Basic Reflection Agent!

In this section, we'll build:

1. `generation_chain`
2. `reflect_chain`



Basic Reflection Agent!

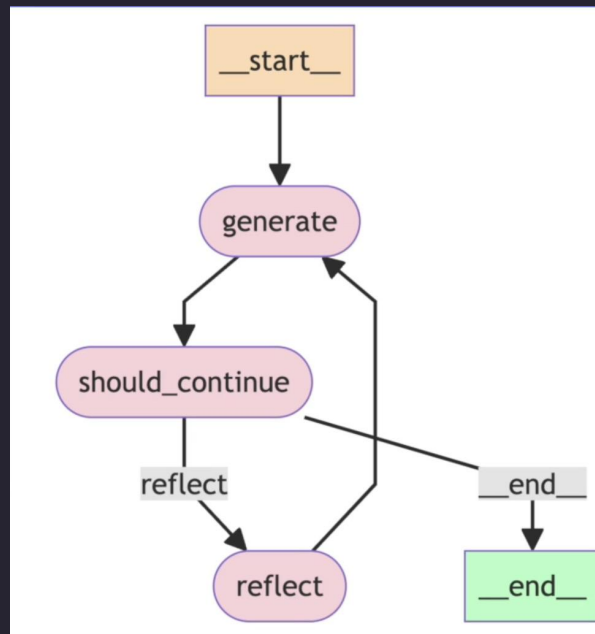
What is a MessageGraph?

It is a class that LangGraph provides that we can use to orchestrate the flow of messages between different nodes

Example use cases: Simple routing decisions, simple chatbot conversation flow

If you just want to pass messages along between nodes, then go for MessageGraph

If the app requires complex state management, we have StateGraph (more on this later)



Basic Reflection Agent!

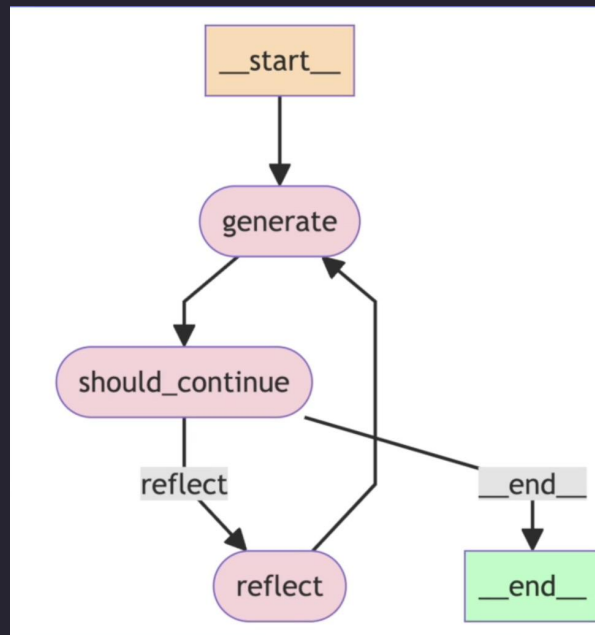
What is a MessageGraph?

To put it simply, MessageGraph maintains a list of messages and decides the flow of those messages between nodes

Every node in MessageGraph receives the full list of previous messages as input

Each node can append new messages to the list and return it

The updated message list is then passed to the next node



Reflexion Agent System

Reflexion Agents in LangGraph

Recap of what we saw previously:

Reflection Agent System consists of a generator and a reflector component

Although, iteratively making a post better is significantly better than just prompting ChatGPT, the content generated is still not grounded in live data

It could be hallucination or outdated content and we have no way of knowing

Reflexion Agent System address this exact drawback

Reflexion Agents in LangGraph

What is Reflexion Agent System:

The reflexion agent, similar to reflection agent, not only critiques it's own responses but also fact checks it with external data by making API calls (Internet Search)

In the Reflection agent pattern, we had to rely on the training data of LLMs but in this case, we're not limited to that.

Reflexion Agents in LangGraph

What is Reflexion Agent System:

The main component of Reflexion Agent System is the "actor"

The "actor" is the main agent that drives everything - it reflects on its responses and re-executes.

It can do this with or without tools to improve based on self-critique that is grounded in external data

Its main sub-components include:

1. Tools/tool execution
2. Initial responder: generate an initial response & self-reflection
3. Revisor: re-respond & reflect based on previous reflections

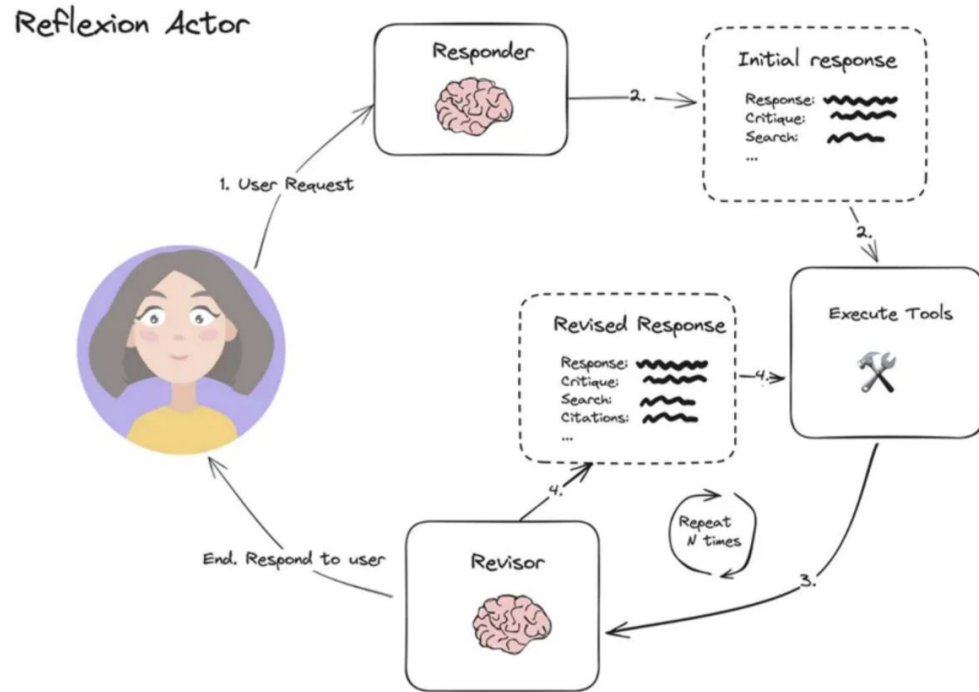
Reflexion Agents in LangGraph

Episodic memory

In the context of Reflexion agents, episodic memory refers to an agent's ability to recall specific past interactions, events, or experiences, rather than just generalized knowledge.

This is crucial for making agents feel more context-aware, personalized, and human-like over time.

Reflexion Agent System



Let's Implement a Reflexion Agent!

LLM Response Parser System

The system converts unstructured LLM outputs into well-defined Python objects through a series of structured parsing steps, ensuring data validation and consistent formatting.

What are the key components?

1. Chat Prompt Template
2. Function Calling with Pydantic Schema
3. Pydantic Parser

LLM Response Parser System

2. Function Calling with Pydantic Schema

Function calling:

Similar to how we make tools available to the LLM, we can also send a schema to the LLM and force it to structure its JSON output according to the schema

Pydantic:

A Python library that defines data structures using classes

Provides automatic validation of JSON data against these class definitions

LLM Response Parser System

3. Pydantic Parser

Takes the JSON output from the LLM's function call

Validates it against the defined Pydantic schema (class definition)

Creates instances of Pydantic classes with the validated data

If the LLMs output does not match with the defined schema, it will throw an error

Re-Act Agent using LangGraph

Re-Act Agent using LangGraph

Think: LLM first thinks about the user prompt/problem

Action: LLM decides if it can answer by itself or if it should use a tool

Action Input: LLM provides the input argument for the tool

Here, langChain executes the tool and returns the output to the LLM

Observe: LLM observe the result of the tool

⋮

Final Answer: "This is your final answer"

ReAct Agent

LangChain:

In LangChain, we used `initialize_agent` as an all-in-one solution.

It combines two key components:

1. `create_react_agent`
2. `AgentExecutor` (We will eliminate the need for this in LangGraph)

ReAct Agent

1. `create_react_agent`: (one that creates the agent)

Takes each tool's name and description

Formats them into a standardized way the LLM can understand

Inserts them into specific placeholders in the ReAct prompt template

It makes the LLM call + takes the LLMs response + parses it

```
Thought: I need to find out when SpaceX's last launch was and calculate days since  
Action: search  
Action Input: "SpaceX most recent launch date"
```

It parses the response of the LLM into one of these two classes: `AgentAction` or `AgentFinish`

Re-Act Agent using LangGraph

AgentAction:

This is a LangChain class that represents an action the agent wants to take. It typically contains:

```
action = AgentAction(  
    tool="calculator",  
    tool_input="2+2",  
    log="I need to calculate the sum of 2 and 2"  
)
```

Re-Act Agent using LangGraph

AgentFinish:

This represents the agent completing its task with a final answer. It typically contains:

```
finish = AgentFinish(  
    return_values={"output": "The answer is 4"},  
    log="I've calculated 2+2 and determined the answer is 4"  
)
```

ReAct Agent

2. AgentExecutor:

Takes the agent from `create_react_agent` and manages the execution loop

Receives the user's question and feeds it to the agent

Identifies which tool to run based on the agent's output (AgentAction or No tool if AgentFinish)

Executes the tool and captures the result

Feeds the result back to the agent for the next decision

Continues this loop until the agent produces an AgentFinish

Returns the final answer to the user

ReAct Agent - LangGraph

Key Advantage:

LangGraph turns the hidden "black box" loop into a visible, editable workflow

You can now add custom nodes, modify the flow, or insert additional logic

Re-Act Agent using LangGraph

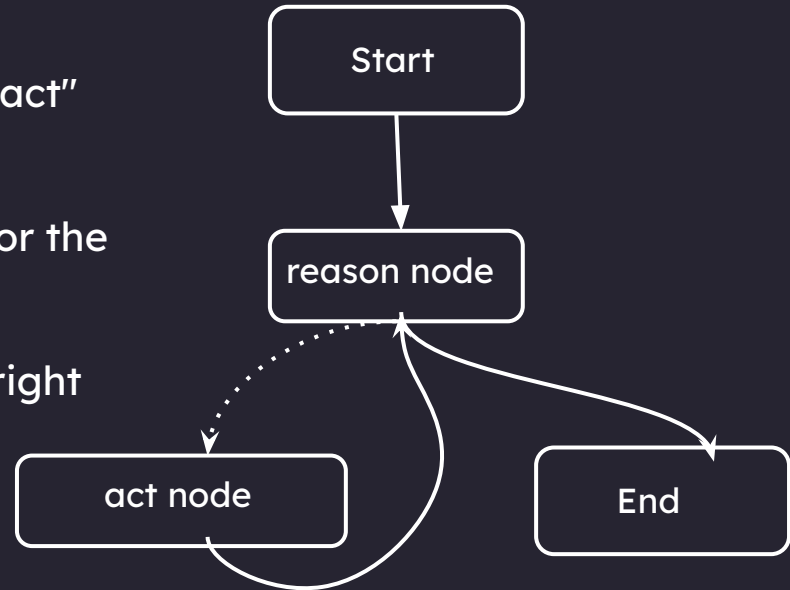
The "reason" node does what create_react_agent did - it thinks and decides

If the reason node outputs an AgentAction, then "act" node executes the tool

Results from the tool flow back to "reason" node for the next decision

When the agent has the final answer, it takes the right path to "end"

This visualization makes the "black box" of AgentExecutor transparent and modifiable



State in LangGraph - Deep Dive

What is State in LangGraph?

State in LangGraph is a way to maintain and track information as an AI system processes data.

Think of it as the system's memory, allowing it to remember and update information as it moves through different stages of a workflow, or graph.

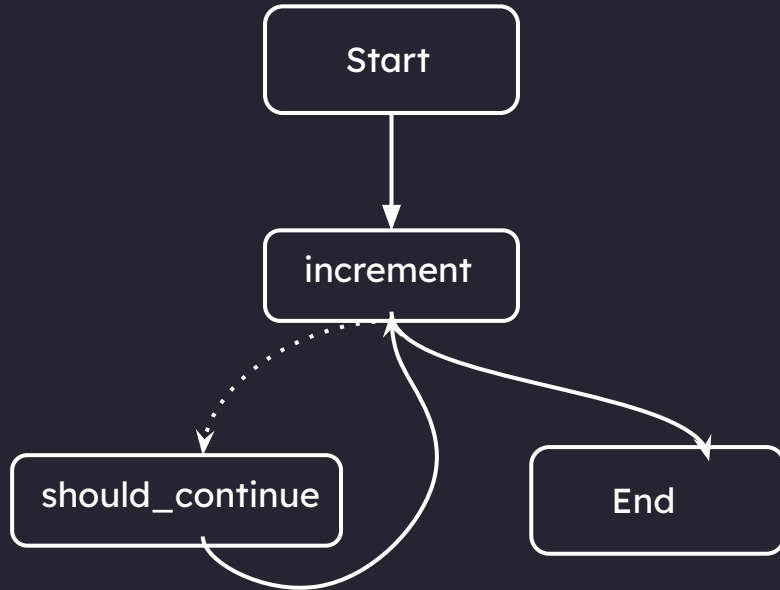
State in LangGraph - Deep Dive

Concepts we will learn:

1. What is StateGraph?
2. Basic State Structures
3. Complex State Structures
4. Manual State Transformation
5. Declarative Annotated State Transformation

State in LangGraph - Deep Dive

1. Basic State Definition



Initial state:

```
{  
  "count": 0  
}
```

ReAct Agent in LangGraph - Deep Dive

States to keep track of:

1. Input that the user provided
2. The parsed output of the LLM response (AgentAction or AgentFinish)
3. The history that has taken place so far

Chatbots using LangGraph

What we'll cover in the next few sections:

1. Basic Chatbot (no memory)
2. Chatbot with Tools
3. Chatbot with memory
4. Chatbot with human-in-loop scenarios
5. Chatbot with more complex state
6. Understanding Time-travel

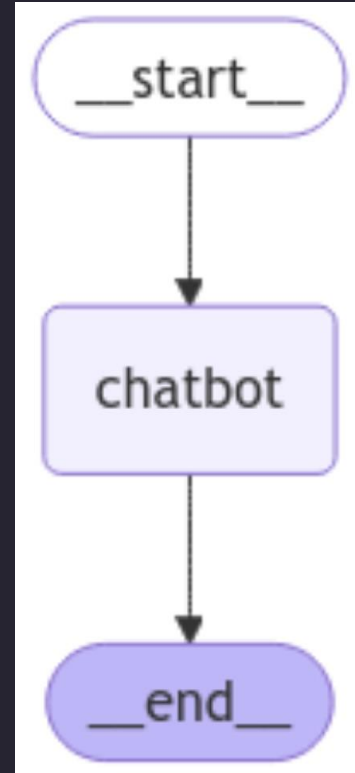
Chat bots using LangGraph

1, Basic Chatbot

- No memory
- No tools

We'll learn:

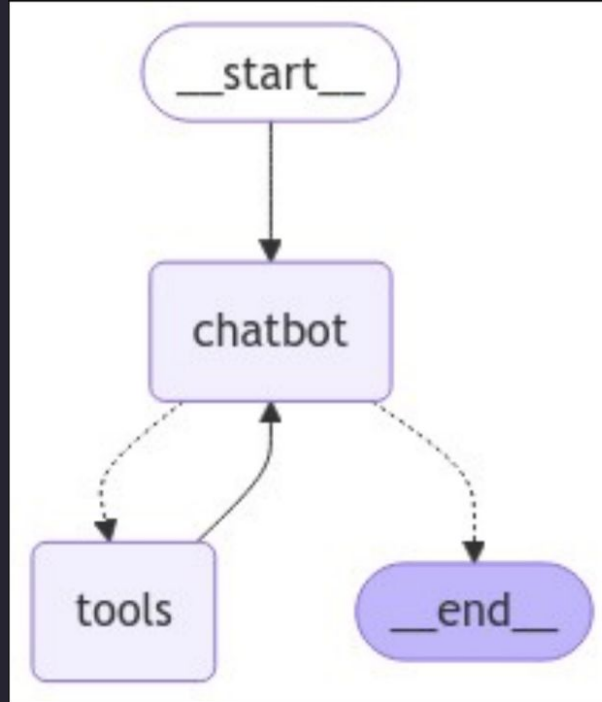
- Graph stream method
- Chat looping
- Use free Llama Model using the Groq interface



Chat bots using LangGraph

2, Chatbot with Tools

- Adding a tool call ability



Memory & Checkpointers

When you build a basic chatbot using LangGraph, you run into an immediate limitation: by default, your chatbot has amnesia.

Every time a user starts a conversation, the bot has no recollection of previous interactions.

This happens because without memory management, each invocation of your graph is completely independent.

This is where the concept of checkpointers in LangGraph come into the picture

Memory & Checkpointers

What is a Checkpointer?

A checkpointer in LangGraph is essentially a way to save the state of your agent or workflow at specific points during execution.

Think of it like saving your progress in a video game. When you reach a checkpoint:

1. The current state of everything is saved
2. If something goes wrong later, you can return to this saved point
3. You don't have to start over from the beginning

Memory & Checkpointers

What is a Checkpointer? (contd.)

In the context of LangGraph nodes and workflows:

- Nodes are the individual steps or components in your workflow
- Checkpoints save the complete state after a node finishes its work
- If an error occurs in a later node, you can resume from the last checkpoint rather than starting the entire workflow again

This is particularly useful for complex workflows where:

- Processing takes significant time or resources
- You want to implement retry mechanisms
- You need persistence across sessions or server restarts

Memory & Checkpointers

Thread ID:

A thread ID is simply a unique identifier for each specific conversation or workflow execution. Think of it like:

- A unique session ID for a user
- A conversation ID that groups related messages together

The thread ID is necessary because:

1. You might have multiple conversations/workflows running simultaneously
2. Each needs its own separate saved state
3. The thread ID helps the system know which saved state belongs to which conversation

Without thread IDs, all your conversations would share the same state, which would cause confusion and errors.

Human In The Loop

A human-in-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.

Use-cases:

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.

Human In The Loop (Design Patterns)

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

1. Approve or Reject:

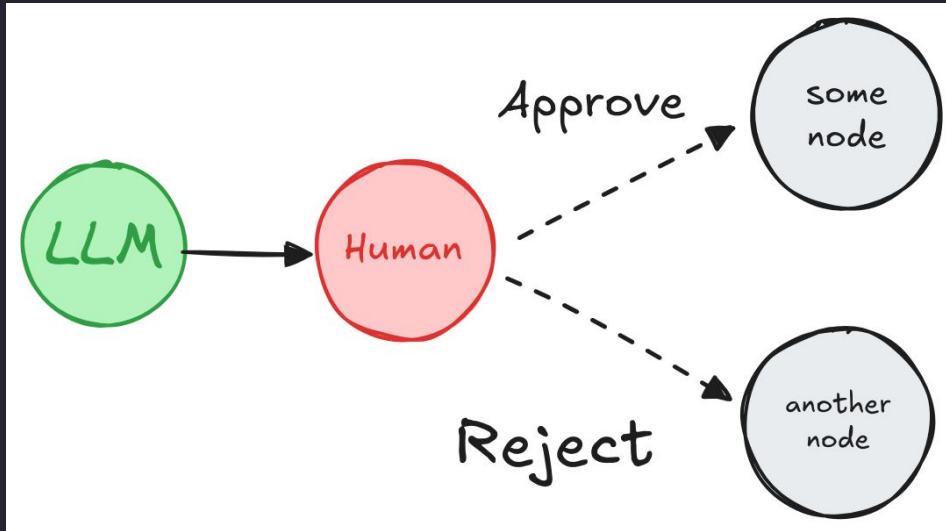
Pause the graph before a critical step, such as an API call, to review and approve the action.

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.

Human In The Loop (Design Patterns)

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

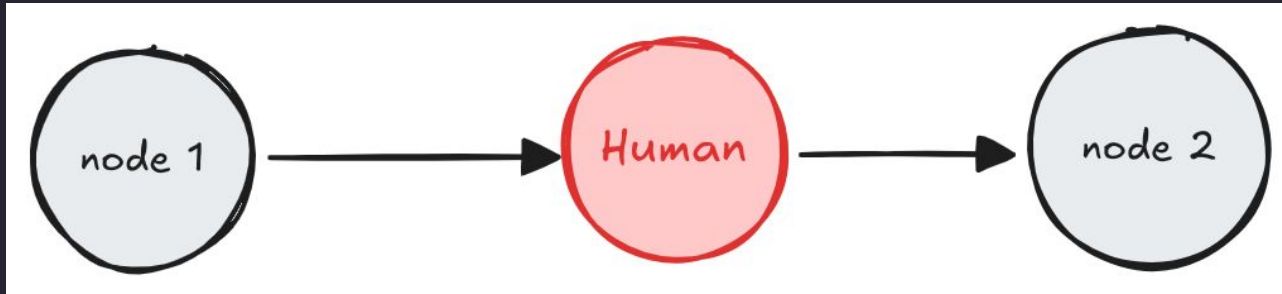
1. Approve or Reject:



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

Human In The Loop (Design Patterns)

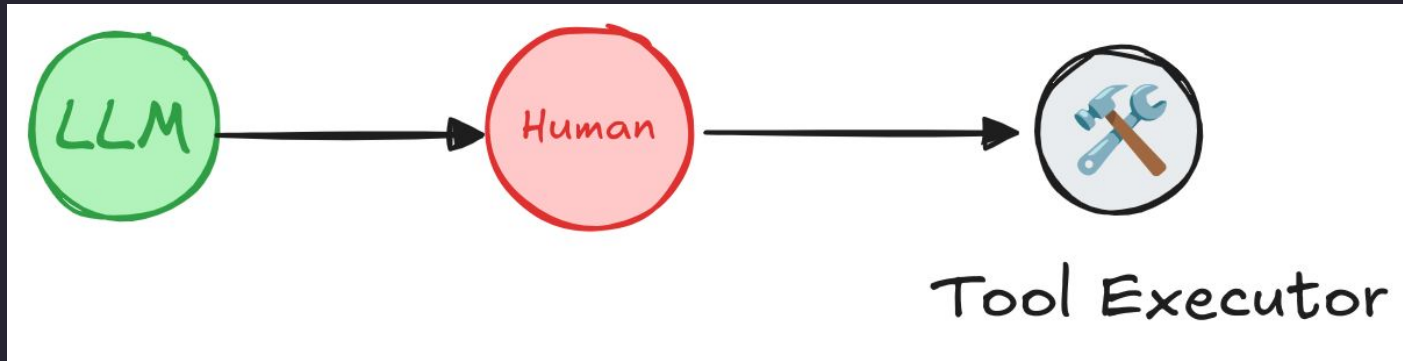
2. Review & Edit State:



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

Human In The Loop (Design Patterns)

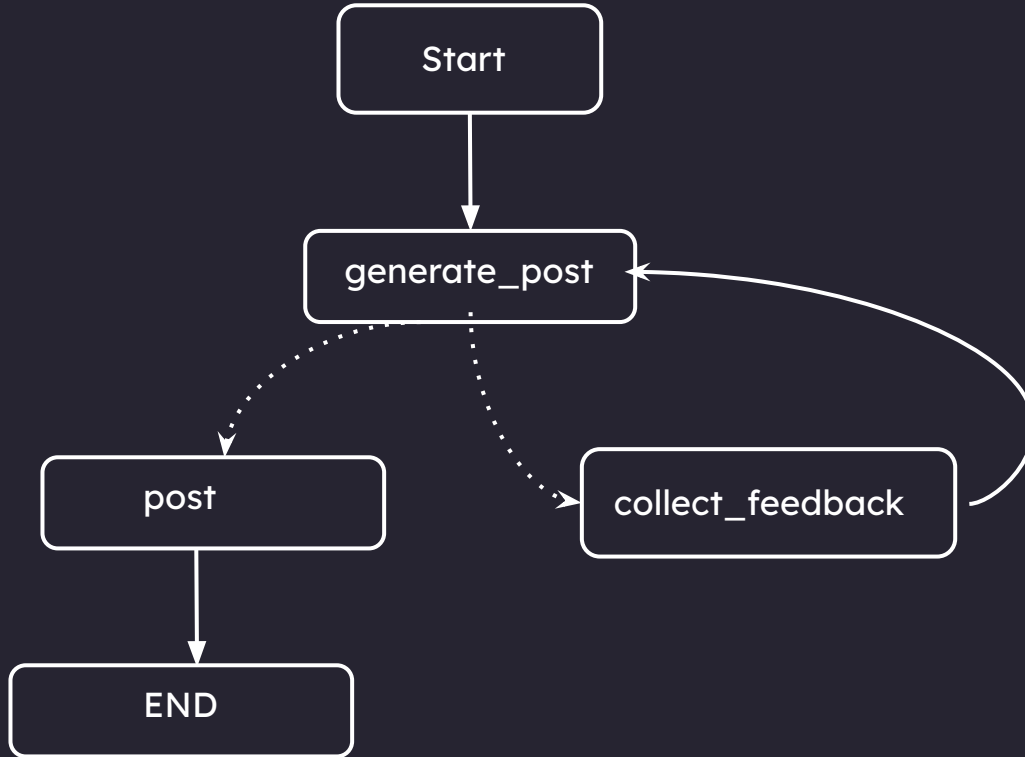
3. Review Tool Calls:



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.

Human In The Loop (Design Patterns)



Human In The Loop

Drawbacks of input():

- Freezes your program completely until someone types something
- Only works in terminals - useless for web apps
- If your program crashes, all progress is lost
- Can only handle one user at a time
- Lives only in your terminal session

This is why we use a special method that LangGraph provides called "interrupt"

Human In The Loop

What is interrupt() & Why Use It?:

- Special LangGraph function that pauses your workflow nicely
- Saves your program's state so it can continue later
- Works in web apps, APIs, and other interfaces
- Handles multiple users/sessions at once
- Survives program crashes and restarts
- Lets humans take their time to respond
- Required for any serious human-in-the-loop system

Human In The Loop

Two ways for using Interrupts:

```
graph = graph_builder.compile(  
    checkpointer = checkpointer,  
    interrupt_before = ["tools"]
```

Interrupt in the compile step

```
human_review = interrupt(  
    {  
        "question": "Is this correct?",  
        "tool_call": tool_call,  
    }  
)  
  
if(review_action == "continue":  
    return Command(goto = "run_tool")
```

Interrupt function with
Command Class

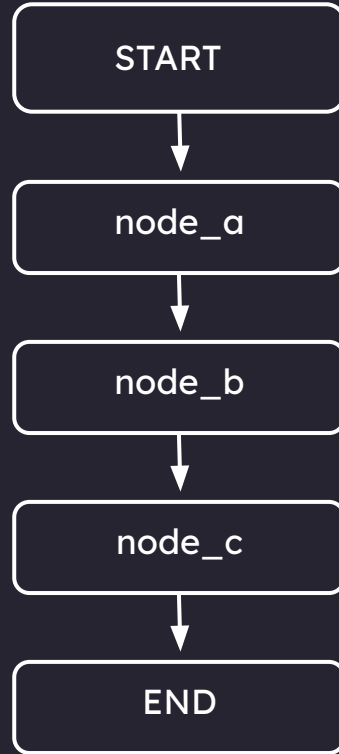
Command Class

The Command class in LangGraph allows us to create edgeless workflows

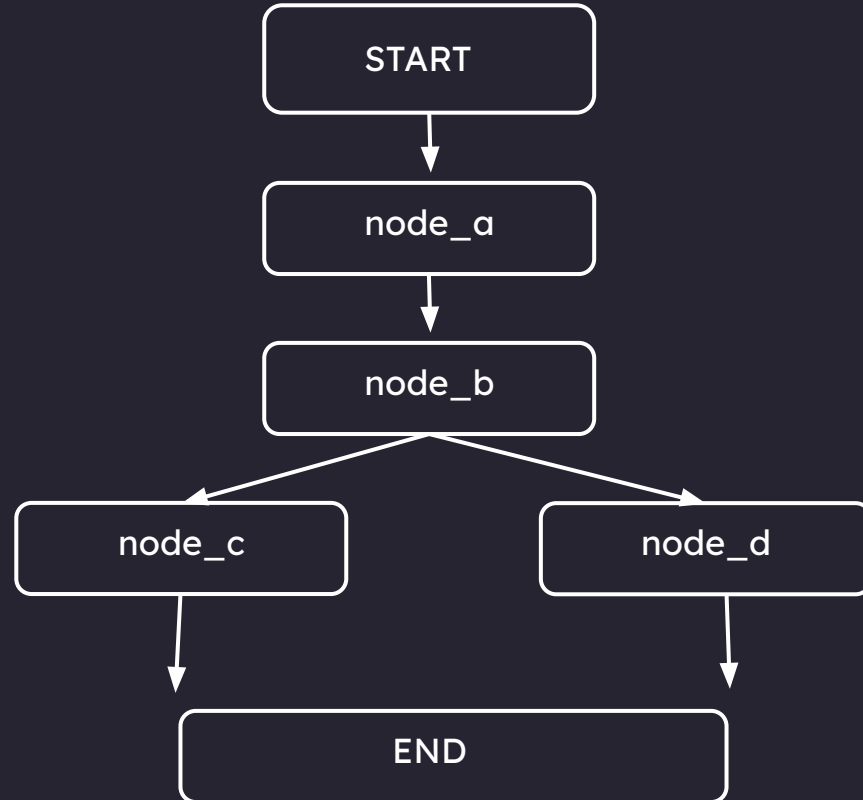


```
def agent(state: MessagesState) -> Command[Literall[...], END]]:  
    ...  
    return Command(  
        goto=..., # The next node(s) to transition to  
        update={"messages": [response]} # Updates to the state  
    )
```

Command Class



Command Class



Interrupts

Operations with Interrupts:

1. **Resume** - Continue execution with input from the user without modifying the state
2. **Update and Resume** - Update the state and then continue execution
3. **Rewind/time Travel** - Go back to a previous checkpoint in the execution
4. **Branch** - Create a new branch from the current execution state to explore alternative paths
5. **Abort** - Cancel the current execution entirely

Each of these operations gives you different ways to control the flow of your graph when it's interrupted

Structured Outputs

It is often useful to have a model return output that matches a specific schema that we define

```
Input: "Tell me a joke about cats"
```

```
Output:
```

```
{  
  'setup': 'Why was the cat sitting on the computer?',  
  'punchline': 'Because it wanted to keep an eye on the mouse!',  
  'rating': 7  
}
```

We have options to get outputs in formats such as - JSON, Dictionary, string, YAML, HTML

Structured Outputs

Pydantic Models for Structured Outputs:

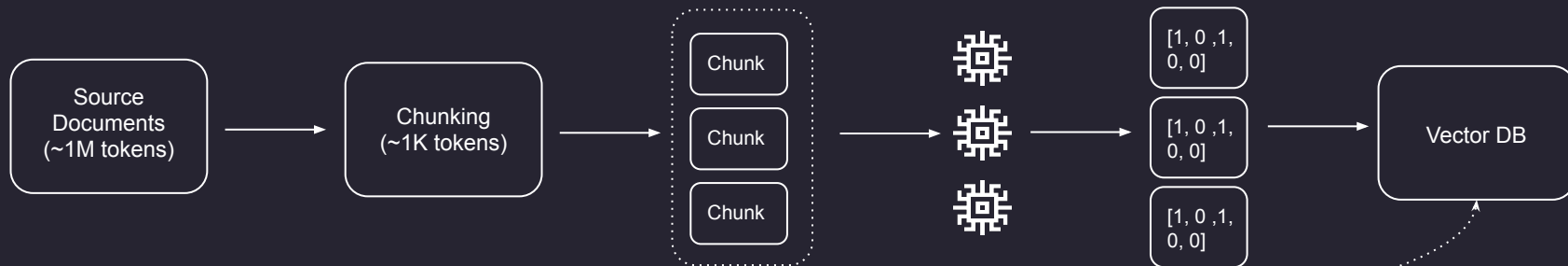
1. Pydantic is a Python library that helps define data structures
2. Acts like a "blueprint" for data
3. Uses Python's type hints (like str, int) to enforce correct data types

How it works in LangChain/LangGraph:

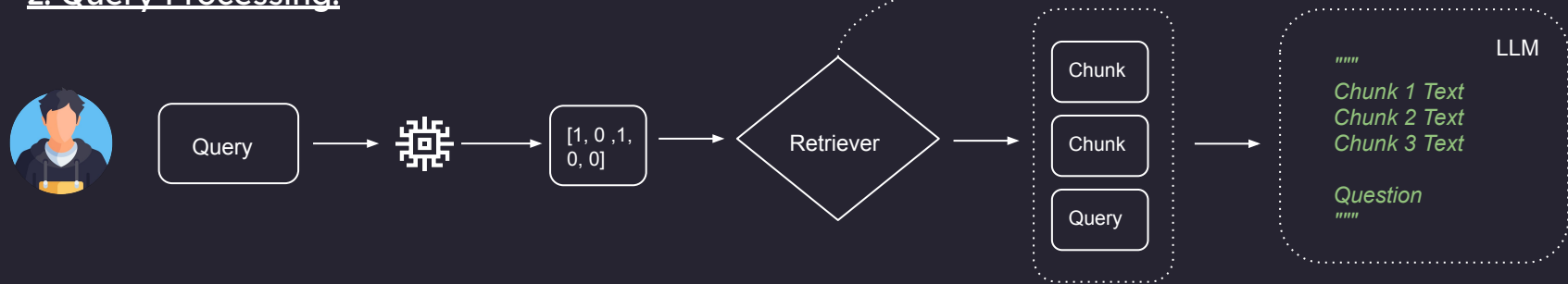
1. Define a class with the fields you need (name, capital, language)
2. Add descriptions to explain what each field means
3. Use `with_structured_output()` to tell the LLM to follow your format

Retrieval Augmented Generation (RAG) System

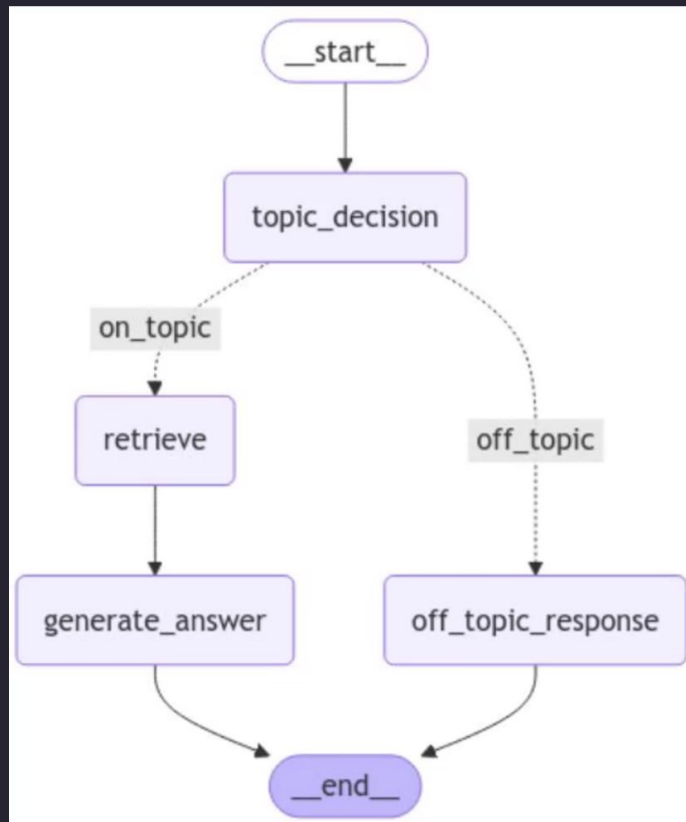
1. Knowledge-base construction:



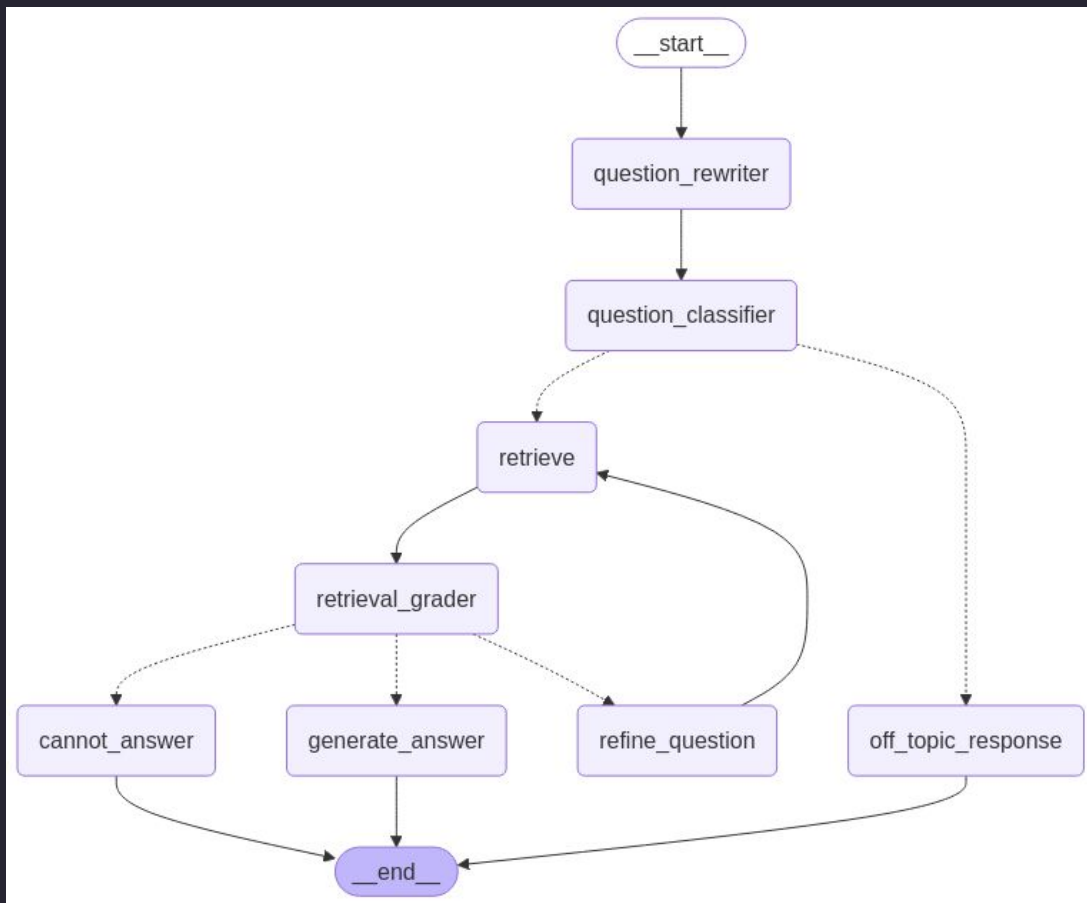
2. Query Processing:



Classification-Driven Retrieval System



Advanced multi-step RAG System



Advanced multi-step RAG System

question_rewriter node significance:

Initial query: "What are Peak Performance Gym's hours?"

Follow-up query: "What about weekends?"

Without rephrasing, the follow-up query lacks context on its own and would likely return irrelevant results if sent directly to a retrieval system.

The rephrasing node transforms "What about weekends?" into "What are Peak Performance Gym's weekend hours?"

Multi-Agent Architectures

We know that an agent is a system that uses an LLM to decide the control flow of an application.

As you develop these systems, they might grow more complex over time, making them harder to manage and scale.

For example, you might run into the following problems:

- Agent has too many tools at its disposal and makes poor decisions about which tool to call next
- context grows too complex for a single agent to keep track of
- there is a need for multiple specialization areas in the system (e.g. planner, researcher, math expert, etc.)

To tackle these, you might consider breaking your application into multiple smaller, independent agents and composing them into a multi-agent system.

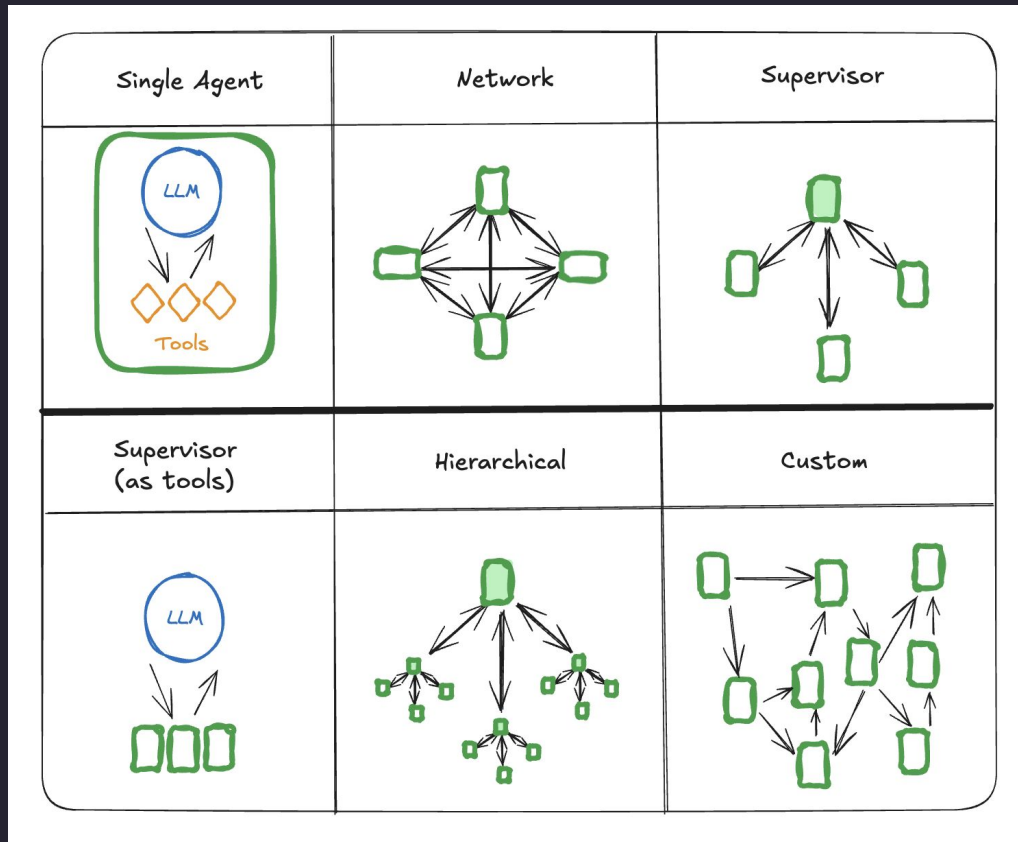
Multi-Agent Architectures

These independent agents can be as simple as a prompt and an LLM call, or as complex as a ReAct agent

The primary benefits of using multi-agent systems are:

- **Modularity:** Separate agents make it easier to develop, test, and maintain agentic systems.
- **Specialization:** You can create expert agents focused on specific domains, which helps with the overall system performance.
- **Control:** You can explicitly control how agents communicate (as opposed to relying on function calling).

Multi-Agent Architectures



Subgraphs

Subgraphs allow you to build complex systems with multiple components that are themselves graphs. A common use case for using subgraphs is building multi-agent systems.

The main question when adding subgraphs is how the parent graph and subgraph communicate, i.e. how they pass the state between each other during the graph execution.

There are two scenarios:

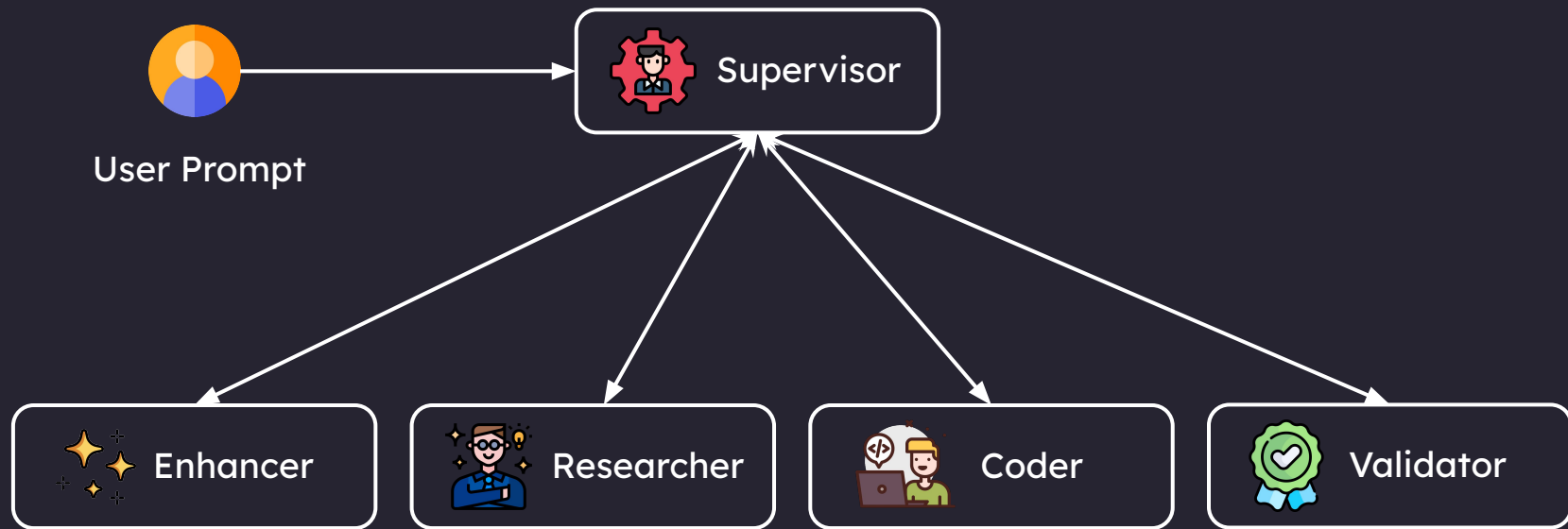
- parent graph and subgraph share schema keys. In this case, you can add a node with the compiled subgraph
- parent graph and subgraph have different schemas. In this case, you have to add a node function that invokes the subgraph: this is useful when the parent graph and the subgraph have different state schemas and you need to transform state before or after calling the subgraph

Supervisor Multi-agent Architecture

In this architecture, we define agents as nodes and add a supervisor node (LLM) that decides which agent nodes should be called next.

We use Command to route execution to the appropriate agent node based on supervisor's decision.

Supervisor Multi-agent Architecture



Streaming in LangGraph

If we're building a responsive app for the users, real-time updates are key to keeping them engaged.

Common use-cases are:

1. Workflow progress (e.g., get state updates after each graph node is executed).
2. LLM tokens as they're generated.
3. Custom updates (e.g., "Fetched 10/100 records")

Streaming in LangGraph

`.stream` and `.astream` are sync and async methods for streaming back outputs from a graph run.

There are several different modes you can specify when calling these methods (e.g. ``graph.stream(input, stream_mode="values")``):

Most common modes are:

1. `stream_mode = "values"`

This streams the full value of the state after each step of the graph.

2. `stream_mode = "updates"`

This streams the updates to the state after each step of the graph

Streaming in LangGraph

stream_mode = "updates" vs stream_mode = "values"

	mode="updates"	mode="values"
node 1	<code>{"messages": ["a"]}</code>	<code>{"messages": ["a"]}</code>
node 2	<code>{"messages": ["b"]}</code>	<code>{"messages": ["a", "b"]}</code>
node 3	<code>{"messages": ["c"]}</code>	<code>{"messages": ["a", "b", "c"]}</code>

Streaming in LangGraph

In prod apps, we usually want to stream more than the state.

In particular, with LLM calls it is common to stream the tokens as they are generated.

We can do this using the ``.astream_events`` method, which streams back events as they happen inside nodes

Each event is a dict with a few keys:

- * ``event``: This is the type of event that is being emitted.
- * ``name``: This is the name of event.
- * ``data``: This is the data associated with the event.
- * ``metadata``: Which contains ``langgraph_node``, the node emitting the event.

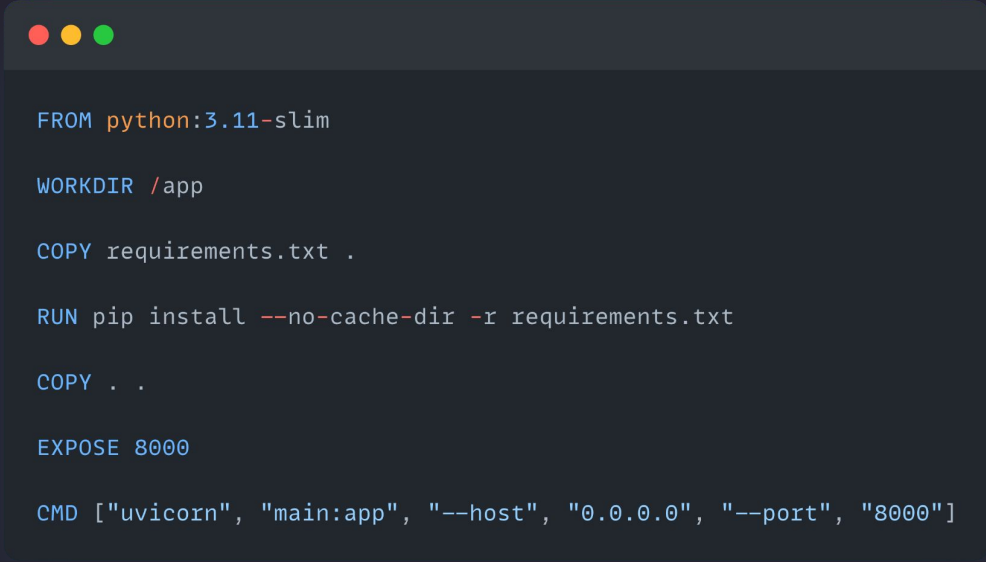
Deployment

Deployment:

So far, we have built an API around the agent

The last step is to deploy our graph using industry-standards.

The standard approach to this is using Docker containers

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the content of a Dockerfile, with each line on a new line and syntax highlighting: FROM is orange, python:3.11-slim is blue, WORKDIR is blue, /app is blue, COPY is blue, requirements.txt is blue, . is blue, RUN is blue, pip install is blue, --no-cache-dir is blue, -r is blue, requirements.txt is blue, CMD is blue, [" is blue, uvicorn is blue, " is blue, main:app is blue, " is blue, --host is blue, " is blue, 0.0.0.0 is blue, " is blue, --port is blue, " is blue, 8000 is blue, and] is blue.

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```


Conclusion

Self-Hosting LangGraph Agents:

TLDR: LangGraph Cloud Optional

1. Build API wrapper around your graph
2. Containerize with Docker
3. Deploy to your preferred cloud provider