# End-to-End LangGraph Course

*By Sunny Savita*

# SYLLABUS INTRODUCTION

- Simple AI Assistant
- RAG (Retrieval augmented generation)
- Chaining with LCEL (LangChain Expression Language)
- Understand Tools and Agents
- Building Tools and Agents from Scratch
- Building Agents using Langchain Class

# SYLLABUS INTRODUCTION

- Graph Structure
  - Understanding about the Graph
  - Direct Acyclic Graph(DAG) vs Cyclic Graph
- What is Langgraph?
- Why Langgraph is Required?
- Creating LangGraph from Scratch
- Creating a LangGraph using inbuilt Classes
- Key concepts and terms in LangGraph
  - Graphs , State, Nodes, Edges,Visualization, Streaming ,Checkpoints, Breakpoints,Configuration, Memory etc.

# SYLLABUS INTRODUCTION

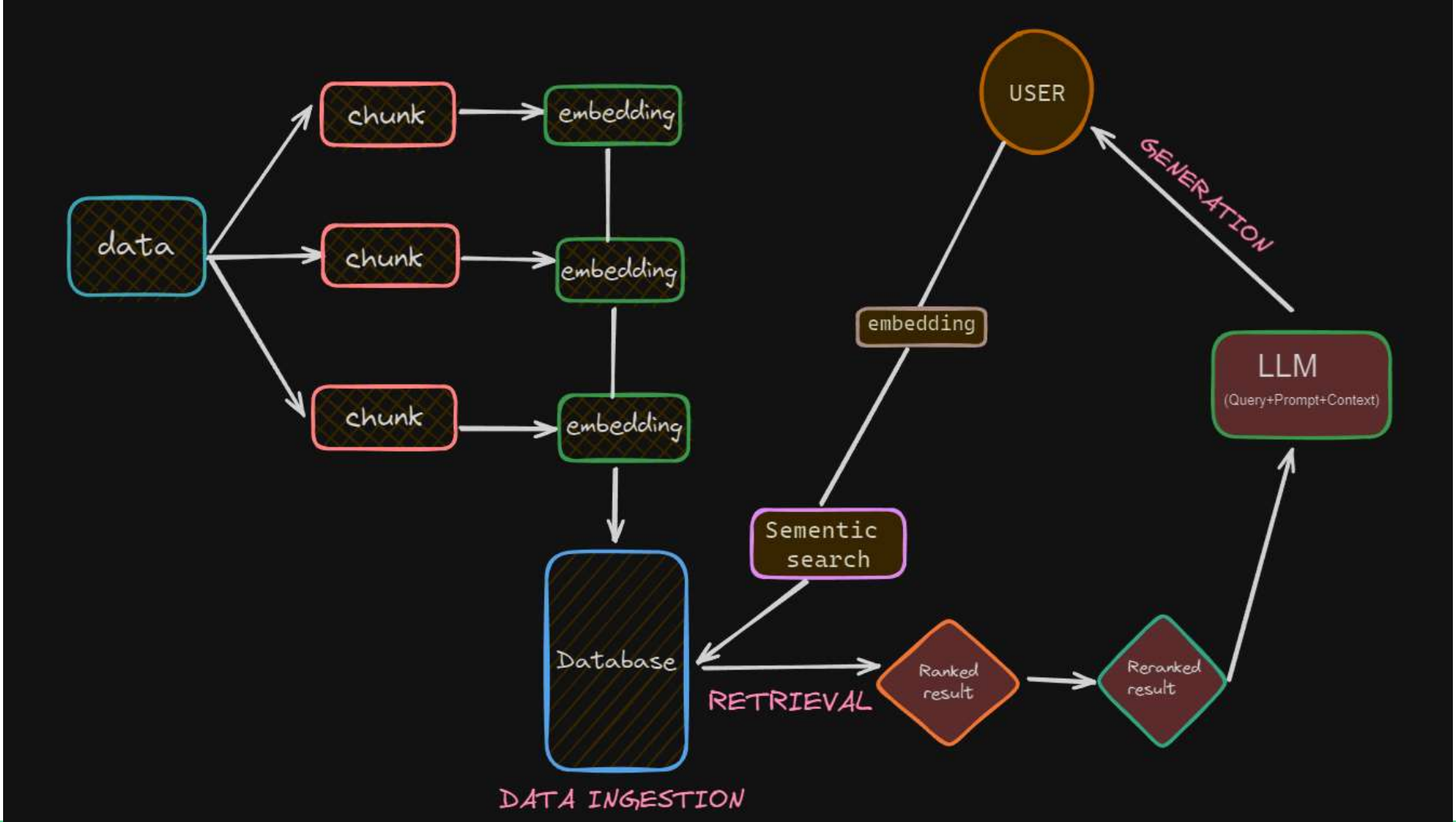
- Creating *CHATBOT* with LangGraph
- Common Agentic Patterns
  - Structure Output
  - Human in Loop
  - ReAct Agent etc.
- Multi-Agent Systems Using LangGraph
- RAGs with LangGraph: CRAG, ARAG, and Self-RAG
- Real-World projects leveraging LangGraph for AI Solutions.

# What is AI Assistant?

- AI assistants LLM model which understand to understand what people say and respond helpfully. They're designed to simulate human conversation and learn from every interaction.

# What is RAG?

Retrieval-Augmented Generation: An AI framework that combines large language models (LLMs) with information retrieval systems. RAG improves the output of LLMs by referencing external knowledge sources, such as web pages, databases, and knowledge bases, to generate responses. This helps to produce more accurate, relevant, and up-to-date text. RAG is useful for chatbots and conversational agents, such as virtual personal assistants and customer support systems.


data
chunk
chunk
chunk
embedding
embedding
embedding
Database
DATA INGESTION
Sementic search
embedding
USER
GENERATION
LLM
(Query+Prompt+Context)
RETRIEVAL
Ranked result
Reranked result

# What is Chaining or Chaining with LCEL?

LangChain Chains connect and orchestrate multiple components

Such as prompts, LLMs, retrieval, output parsers.

LCEL makes it easy to build complex chains from basic components, and supports out of the box functionality such as streaming, parallelism, and logging.

Chains that are built with LCEL. In this case and Chains constructed by subclassing from a legacy Chain class.

Chains lets you create a predefined sequence of tools

# What is Agent?

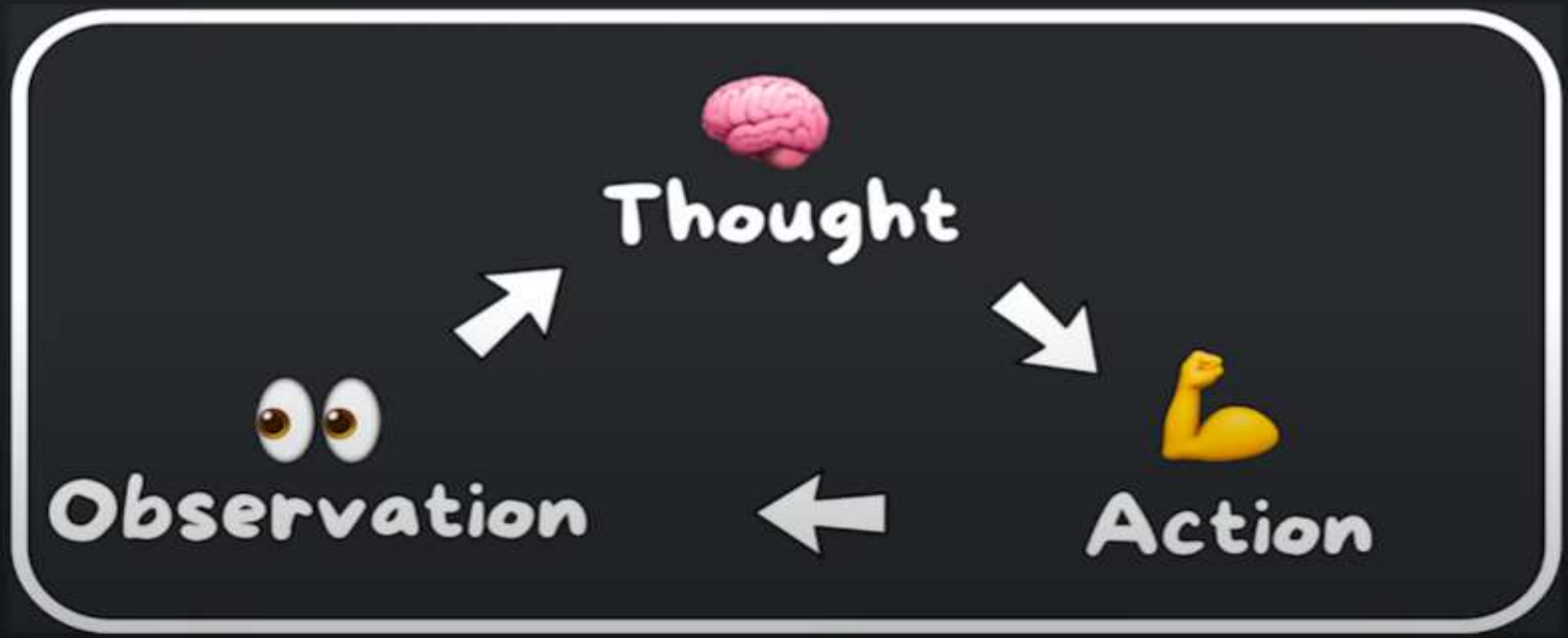
The core idea of agents is to use a language model to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

And as seen below, the agent creates a chain in **real-time**, **reflects** on the question, and goes through a process of **action**, **observation**, **thought**, **action**, **observation, thought**...until the final answer is reached.

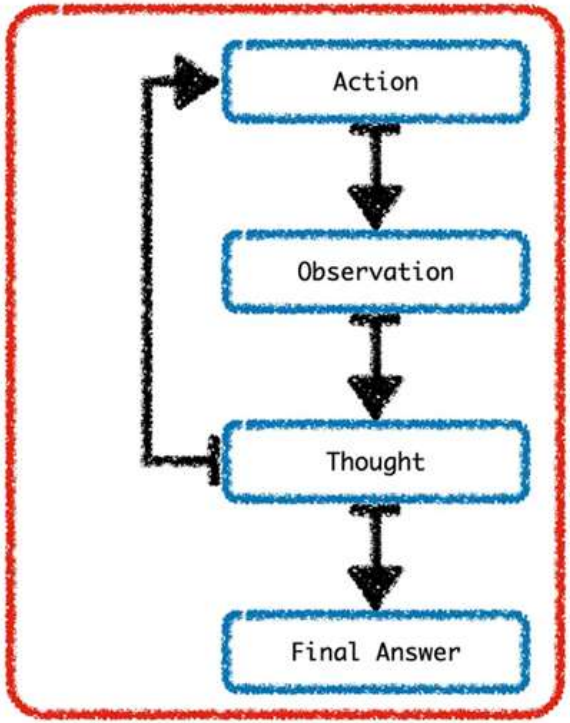
The agent is then executed using an Agent Executor , which manages the interaction between the agent and the tools.

# Plan-and-execute

## Action Agent Control Flow



# LangChain Agent - Sequence Of Events

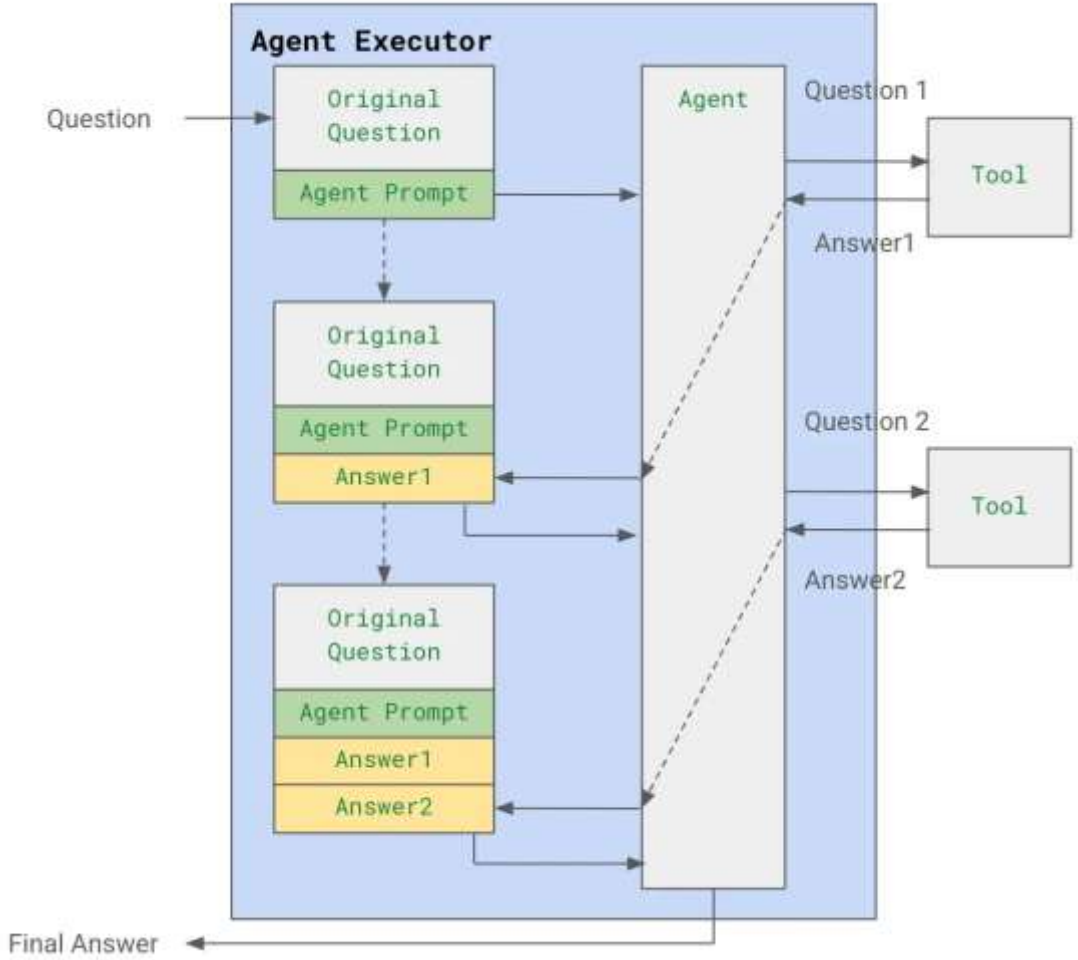


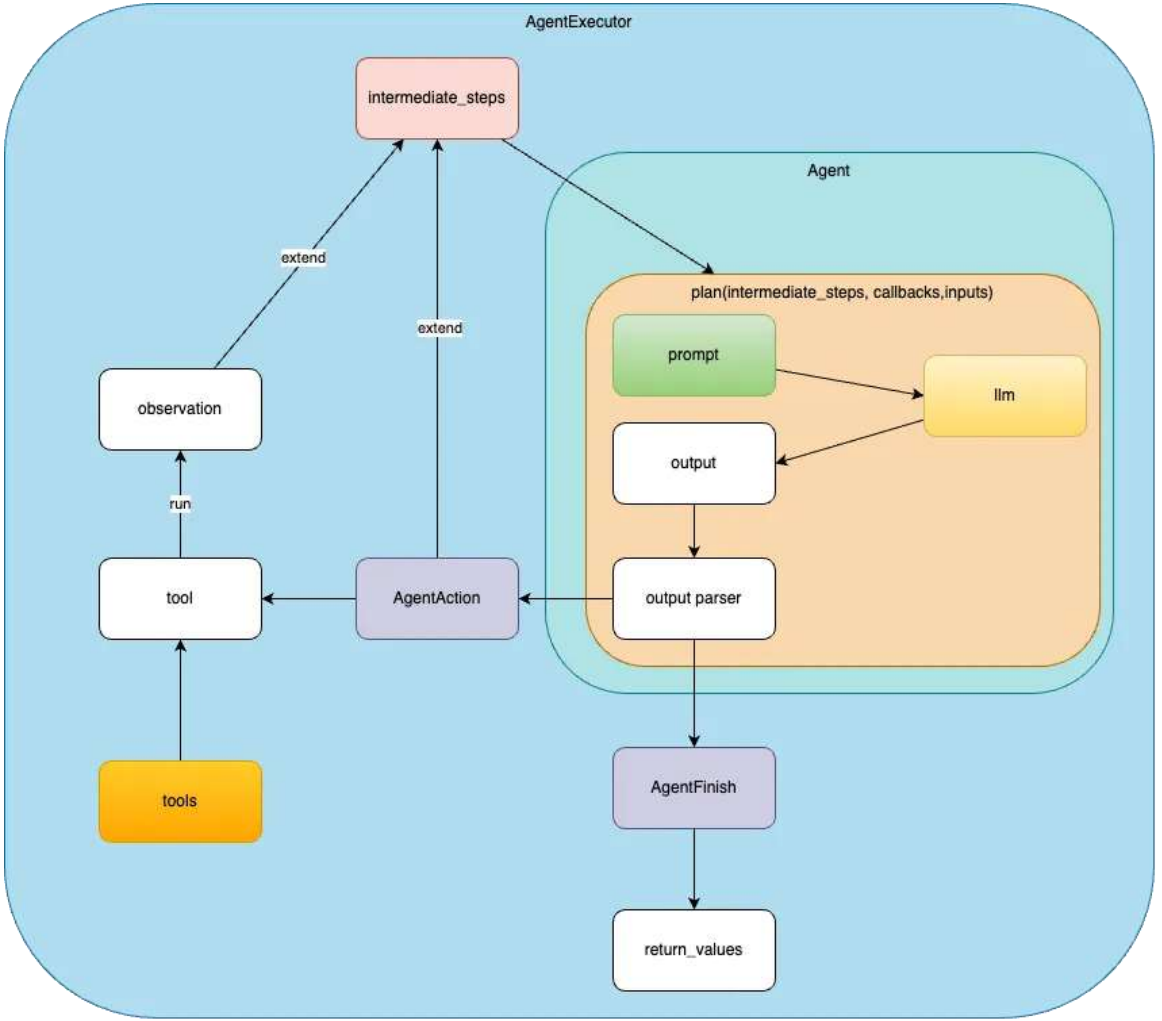
# LangChain output parsing works with prompt templates

```
EXAMPLES = ["""
Question: What is the elevation range
for the area that the eastern sector
of the Colorado orogeny extends into?

Thought: I need to search Colorado orogeny, find
the area that the eastern sector of the Colorado
orogeny extends into, then find the elevation range
of the area.

Action: Search[Colorado orogeny]

Observation: The Colorado orogeny was an
episode of mountain building (an orogeny) in
Colorado and surrounding areas.

Thought: It does not mention the eastern sector.
So I need to look up eastern sector.
Action: Lookup[eastern sector]

...

Thought: High Plains rise in elevation from
around 1,800 to 7,000 ft, so the answer is 1,800 to
7,000 ft.

Action: Finish[1,800 to 7,000 ft]""",
]
```

LangChain library functions parse the LLM's output assuming that it will use certain keywords.

Example here uses **Thought**, **Action**, **Observation** as keywords for Chain-of-Thought Reasoning. (ReAct)


Agent Executor
Question
Original Question
Agent Prompt
Agent
Question 1
Tool
Answer1
Original Question
Agent Prompt
Answer1
Question 2
Tool
Answer2
Original Question
Agent Prompt
Answer1
Answer2
Final Answer

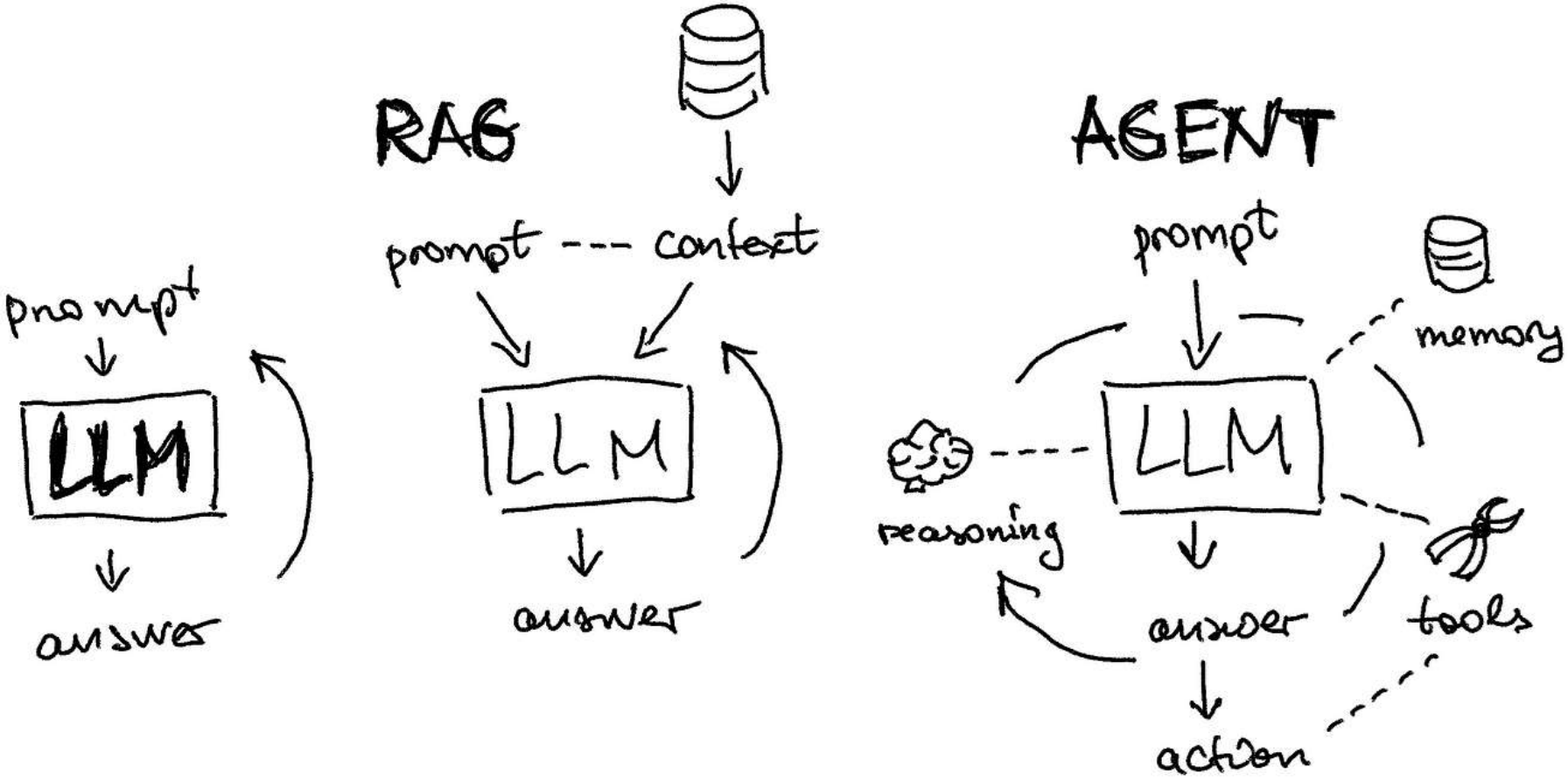

AgentExecutor
intermediate_steps
Agent
extend
extend
plan(intermediate_steps, callbacks,inputs)
prompt
llm
observation
output
run
tool
AgentAction
output parser
tools
AgentFinish
return_values

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
  observation = run(next_action)
  next_action = agent.get_action(..., next_action, observation)
return next_action
```

# What is Tools?

In LangChain, an “Agent” is an AI entity that interacts with various “Tools” to perform tasks or answer queries. Tools are essentially functions that extend the agent's capabilities by allowing it to perform specific actions, like retrieving the current time or accessing an external database.

Agents let the model use tools in a loop, so that it can decide how many times to use tools.


prompt
LLM
answer
RAG
prompt --- context
LLM
answer
AGENT
prompt
memory
reasoning
LLM
answer
tools
action

# Types of Agents or Agentic Patterns

1. Tool Calling
2. Structure Output
3. Human in Loop
4. Map- Reduce
5. MultiAgents
6. Planning
7. Reflection(Reflex Agents)
8. ReAct Agent(Learning Agents)
9. Hierarchical Agents

# Benefits of AI Agents

1. Enhanced Efficiency and Productivity
2. Improved Decision-Making
3. 24/7 Availability and Scalability
4. Personalized Customer Experiences
5. Cost Reduction and Increased Revenue
6. Innovation and New Opportunities

# AI Agent Applications Domain

1. Healthcare
2. Education
3. E-Commerce and Retail
4. Finance
5. Manufacturing
6. Marketing

## Product Examples of Artificial Intelligent Agents

1. Intelligent Personal Assistants
2. Chatbots
3. Autonomous Robots
4. Game Playing Agents
5. Fraud Detection Agents

# Built-in AI Agent Framework  or Platform

1. AutoGen
2. Crewai
3. PhiData
4. Cogniflow
5. LangChain
6. Llama-Index
7. Vertex AI Agent Builder
8. D-iD etc...

# Famous Built In project on Top of AI Agents

1. BabyAGI
2. Autogpt
3. MetaGPT
4. ChatDev
5. JARVIS
6. OpenDevin etc...

# Introduction of Graph

Graph is a collection of edges and Nodes

**Nodes (or vertices)**: Represent the objects or entities.

**Edges (or links)**: Represent the connections or relationships between the nodes.

**Directed**: Edges have a direction (like an arrow), indicating a one-way relationship.

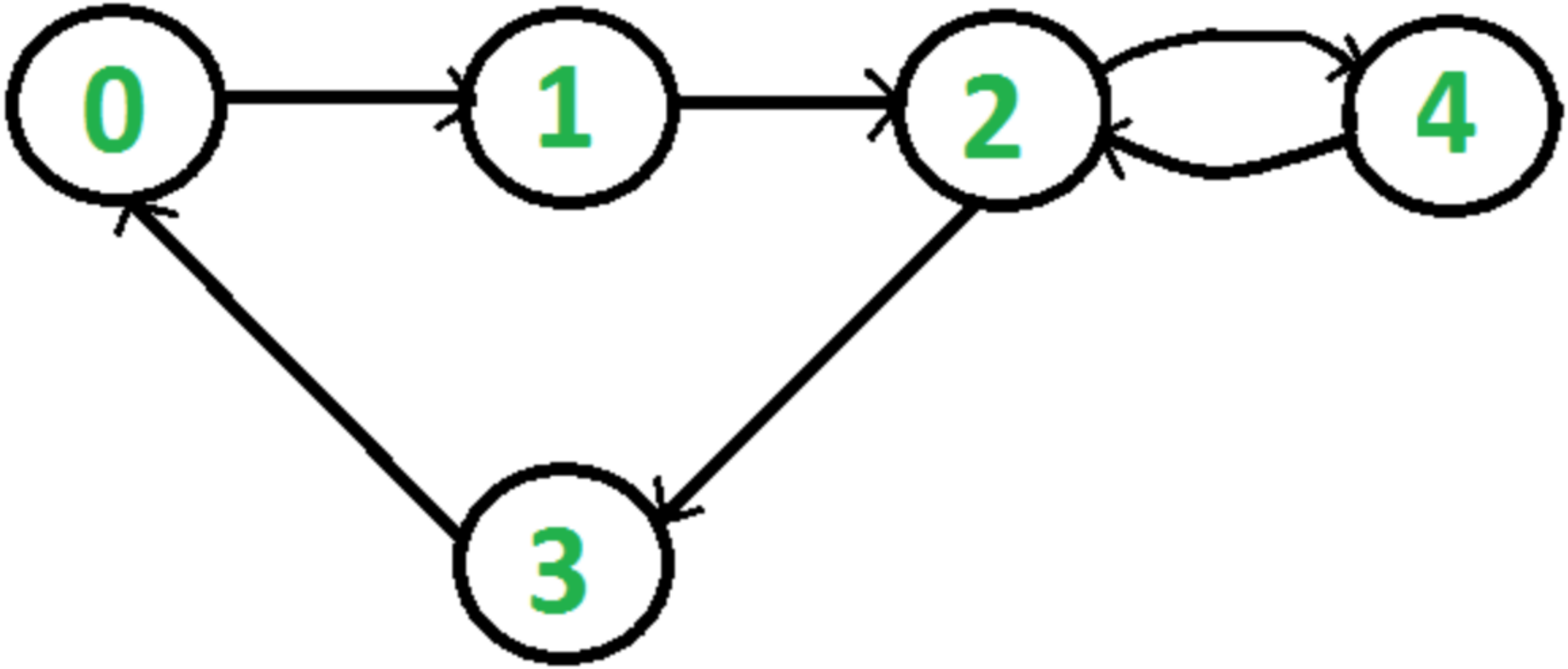
# Introduction of Graph

A **directed cyclic graph** (DCG) and a **directed acyclic graph** (DAG) are types of graphs that describe relationships between nodes, but they differ in how cycles (loops) are handled:

**1. Directed Cyclic Graph (DCG):**

- **Directed**: Each edge has a direction, indicating a one-way relationship between nodes.
- **Cyclic**: Contains at least one cycle, meaning you can start at a node, follow directed edges, and eventually return to the same node.

**Example**: In a DCG, if node A points to node B, and node B points back to node A, there's a cycle.


0
1
2
4
3

# Introduction of Graph

## 2. Directed Acyclic Graph (DAG):

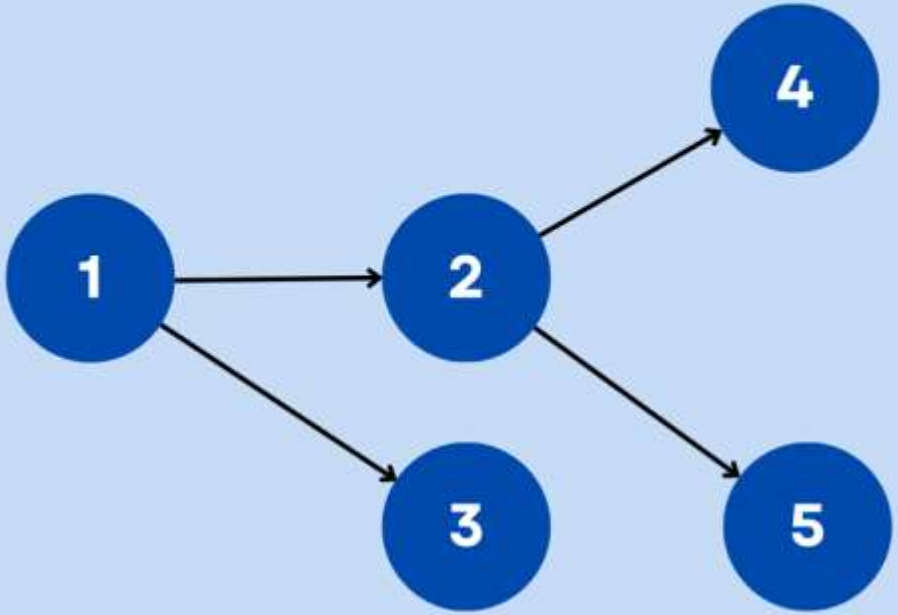
- **Directed**: Each edge has a direction.
- **Acyclic**: Contains no cycles, meaning there's no way to start at a node and follow directed edges to return to that same node.

**Example**: A DAG is often used in workflows (like task scheduling), where each task must follow a certain sequence without repeating any step.

**Key Difference**:

- A **DCG** allows circular paths, while a **DAG** strictly forbids them.

# Directed Acyclic Graph



# What is Langraph?

LangGraph is an advanced library built on top of LangChain, designed to enhance your Large Language Model (LLM) applications by introducing cyclic computational capabilities.

Lang Graph is a module built on top of LangChain to better enable creation of cyclical graphs, often needed for agent runtimes.

LangGraph introduces the ability to add cycles, enabling more complex, agent-like behaviors where you can call an LLM in a loop, asking it what action to take next.

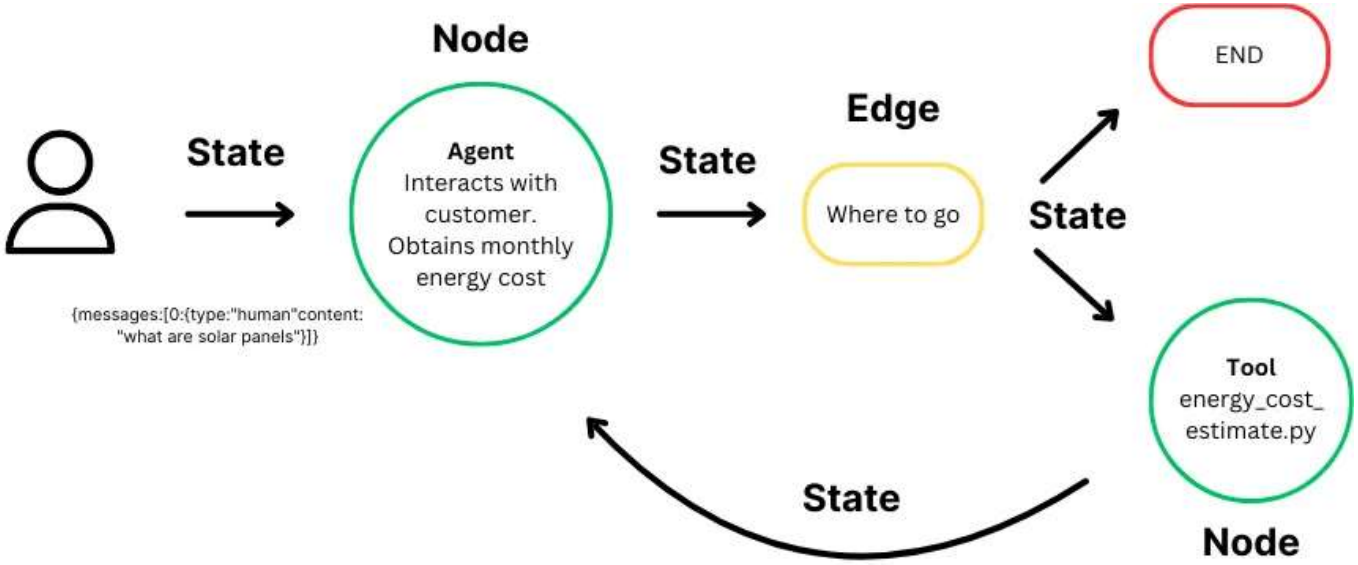
While LangChain allows the creation of Directed Acyclic Graphs (DAGs) for linear workflows, LangGraph takes this a step further by enabling the addition of cycles,

Which are essential for developing complex, agent-like behaviors. These behaviors allow LLMs to continuously loop through a process, dynamically deciding what action to take next based on evolving conditions.

# What is Langraph?

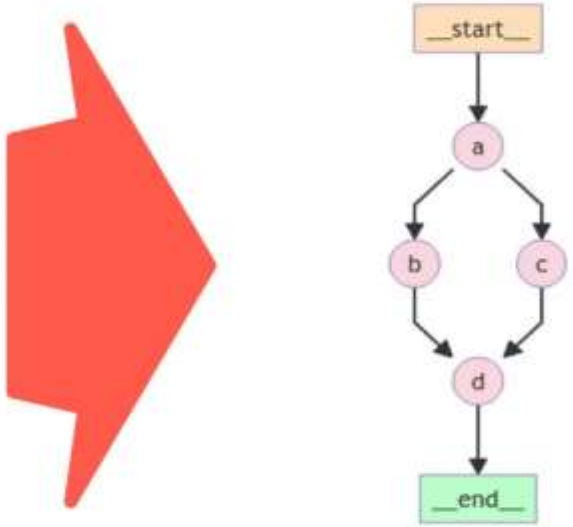
The aim of LangGraph is to have level of control when it comes to executing autonomous AI agents.

# What is Langraph?



# LangGraph Example

```
builder = StateGraph(State)
builder.add_node("a", ReturnNodeValue("I'm A"))
builder.set_entry_point("a")
builder.add_node("b", ReturnNodeValue("I'm B"))
builder.add_node("c", ReturnNodeValue("I'm C"))
builder.add_node("d", ReturnNodeValue("I'm D"))
builder.add_edge("a", "b")
builder.add_edge("a", "c")
builder.add_edge("b", "d")
builder.add_edge("c", "d")
builder.set_finish_point("d")
graph = builder.compile()
```



# Why is Langraph?

LangGraph is framework-agnostic, with each node functioning as a regular Python function.It extends the core Runnable API (a shared interface for streaming, async, and batch calls) to facilitate:

Seamless state management across multiple conversation turns or tool usages.

Flexible routing between nodes based on dynamic criteria

Smooth transitions between LLMs and human intervention

Persistence for long-running, multi-session applications

# LangGraph vs LangChain Agents

LangGraph is an orchestration framework for complex agentic systems and is more low-level and controllable than LangChain agents. On the other hand, LangChain provides a standard interface (LangChain Agents in their Previous Version)to interact with models and other components, to Automate the Flow.

# Important Terms of Langgraph

**Graphs:** At its core, LangGraph models agent workflows as graphs. You define the behavior of your agents using three key components:

1. **State:** A shared data structure that represents the current snapshot of your application. It can be any Python type, but is typically a Type Dict or Pydantic Base Model.
2. **Nodes:** Python functions that encode the logic of your agents. They receive the current State as input, perform some computation or side-effect, and return an updated State.
3. **Edges:** Python functions that determine which Node to execute next based on the current State. They can be conditional branches or fixed transitions.

# Important Terms of Langgraph

Edges define how the logic is routed and how the graph decides to stop. This is a big part of how your agents work and how different nodes communicate with each other. There are a few key types of edges:

- **Normal Edges:** Go directly from one node to the next.
- **Conditional Edges:** Call a function to determine which node(s) to go to next.
- **Entry Point:** Which node to call first when user input arrives.
- **Conditional Entry Point:** Call a function to determine which node(s) to call first when user input arrives.

# Important Terms of Langgraph

**START Node**

The START Node is a special node that represents the node sends user input to the graph. The main purpose for referencing this node is to determine which nodes should be called first.

```
from langgraph.graph import START

graph.add_edge(START, "node_a")
```

**END Node**

The END Node is a special node that represents a terminal node. This node is referenced when you want to denote which edges have no actions after they are done.

```
from langgraph.graph import END

graph.add_edge("node_a", END)
```

# Important Terms of Langgraph

**StateGraph:** The StateGraph class is the main graph class to uses. This is parameterized by a user defined State object.

**Message Graph:** The Message Graph class is a special type of graph. The State of a Message Graph is ONLY a list of messages. This class is rarely used except for chatbots, as most applications require the State to be more complex than a list of messages.

**Compiling your graph:** To build your graph, you first define the state, you then add nodes and edges, and then you compile it.

```
graph = graph_builder.compile(...)
```

# Important Terms of Langgraph

## Visualization

It's often nice to be able to visualize graphs, especially as they get more complex. LangGraph comes with several built-in ways to visualize graphs. See this how-to guide for more info.

## Streaming

LangGraph is built with first class support for streaming. There are several different ways to stream back results

## Checkpointer

LangGraph has a built-in persistence layer, implemented through checkpointers. When you use a checkpointer with a graph, you can interact with the state of that graph. When you use a checkpointer with a graph, you can interact with and manage the graph's state. The checkpointer saves a checkpoint of the graph state at every super-step, enabling several powerful capabilities:

## Configuration

When creating a graph, you can also mark that certain parts of the graph are configurable. This is commonly done to enable easily switching between models or system prompts. This allows you to create a single "cognitive architecture" (the graph) but have multiple different instance of it.

“The best way to predict the future is to create it.”

- Sunny Savita