Building Advanced AI Agents for Workflow Automation with LangGraph

Yogesh Haribhau Kulkarni

YHK

Outline

Introduction to LangGraph

Objectives for learning LangGraph

- Build powerful AI agents for products and business applications
- Learn to create agents that think and make decisions independently
- Master human-in-the-loop approval processes
- Practical implementation for job opportunities
- Foundation for building autonomous AI systems

YHK

Prerequisites

- Python Programming: Essential for implementation
- LangChain Understanding: Foundation for LangGraph concepts
- Chat Models: Core LLM interaction patterns
- Prompt Templates: Structured input formatting
- RAG Systems: Retrieval-augmented generation knowledge
- Agents and Tools: Basic agent architecture understanding

YHK

Background

(Ref: LangGraph Crash Course - Harish Neel)

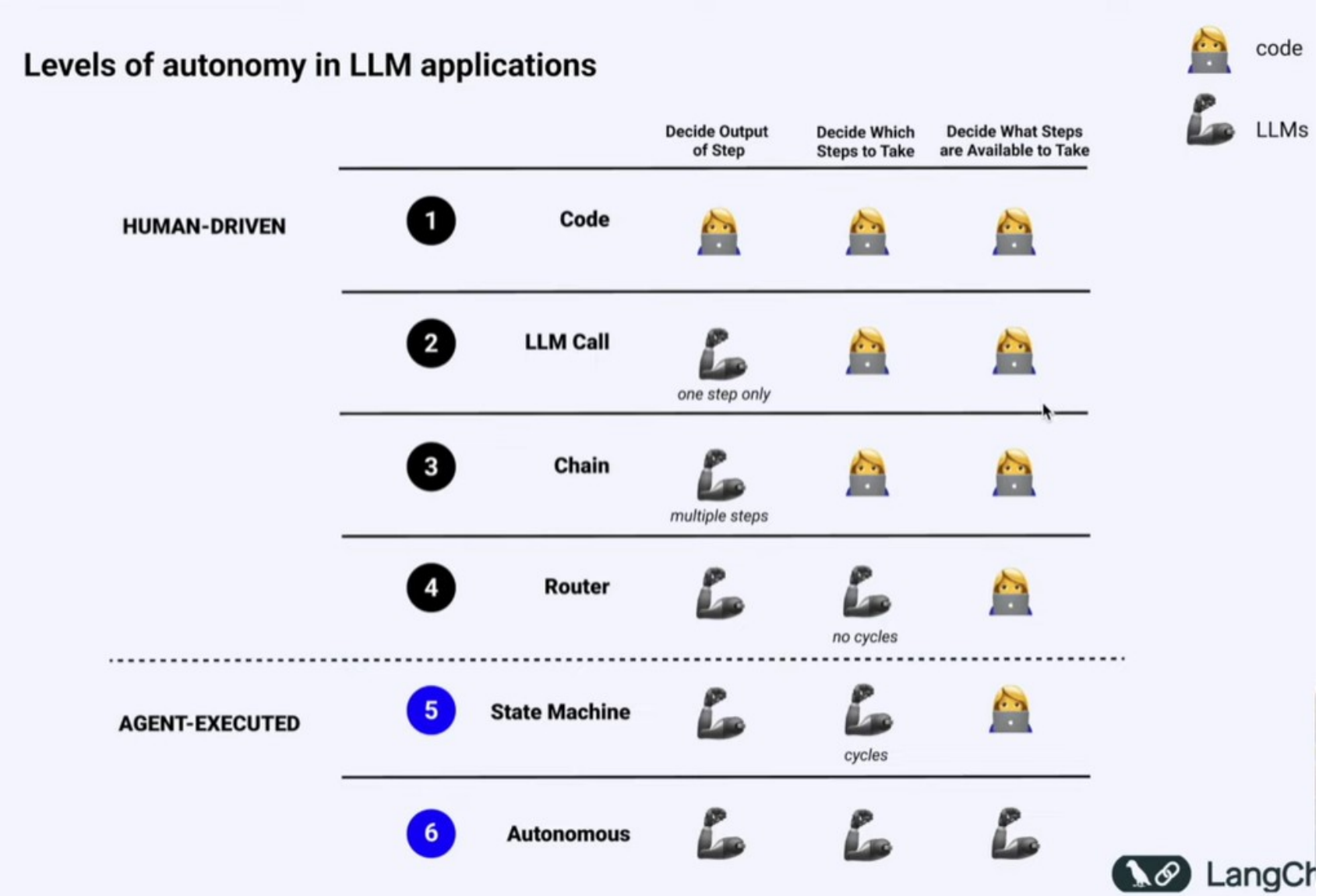
YHK

Levels of Autonomy in LLM Applications

- Code: Very little freedom, 100% deterministic, executes exactly what we tell it. So, need to write for all cases.
- LLM call: One task, atomic, limited decision-making capabilities.
- Chains: unidirectional sequence of operations, Some autonomous reasoning, but can't branch out, LLM call at each position
- Routers: unidirectional, branching based on reasoning, chains in each branch
- Agents (State Machine): High autonomy with independent thinking, Router with Loops, any direction, Human-in approval, adv memory, refers history
- LangGraph Position: Enables maximum agent autonomy
- Progressive Evolution: Understanding the journey from code to agents
- Freedom Spectrum: From rigid execution to flexible decision-making

YHK

Levels of Autonomy in LLM Applications



Introduction to AI Agents

- AI agents are the problem solvers of the AI world
- They are capable of autonomous thinking and decision-making
- Unlike chains and routers, agents don't just follow specific instructions
- Agents can decide for themselves what steps to take
- They represent AI systems that can make independent decisions
- Agents take problem-solving capabilities a step further than traditional AI systems

YHK

Understanding Tools for AI Agents

- Tools are specific functions that agents use to complete tasks
- Analogy: Like a chef's kitchen tools - knife for cutting, oven for baking, blender for mixing
- Tools provide special abilities to AI agents
- Common tools include calculator, search engine, and calendar functions
- Tools extend the agent's capabilities beyond basic reasoning
- They enable agents to interact with external systems and perform specific operations

YHK

The ReAct Agent Pattern

- ReAct stands for Reasoning plus Acting
- One of the best-known patterns for building AI agents today
- Mimics how human beings think and solve problems
- Combines reasoning capabilities with action execution
- Provides a structured approach to agent decision-making
- Forms the foundation for many modern AI agent implementations

YHK

ReAct Pattern Components

- **Think**: LLM first thinks about the user's prompt or problem
- **Action**: LLM decides if it can answer directly or needs a tool
- **Action Input**: LLM provides input arguments for the selected tool
- **Observe**: LLM observes the output of the tool execution
- Cycle continues until the final answer is reached
- Mirrors human problem-solving approach: think $\rightarrow$ act $\rightarrow$ observe

YHK

ReAct Pattern Workflow

- Human problem-solving mimicry: identify problem, determine action, observe results
- If answer is found, the cycle ends
- If answer is not found, the cycle repeats with new thinking
- Control flow: LLM suggests $\rightarrow$ System executes $\rightarrow$ Results return to LLM
- LLM maintains complete context of all previous think-action-observation cycles
- Multi-step problems require multiple cycles until resolution

YHK

Control Flow in ReAct Pattern

- LLM acts as the reasoning component but doesn't execute tools directly
- LangChain system handles the actual tool execution
- LLM provides arguments, system executes with those arguments
- Tool execution output is returned to the LLM for observation
- LLM has full context of all previous interactions
- System manages the orchestration between reasoning and action

YHK

Agent Architecture: Brain + Tools

- LLM provides the reasoning ability (the "brain")
- Tools provide execution capabilities (API calls, Google search, Python functions)
- Combination of brain + tools = AI Agent
- Reasoning ability alone is not sufficient for complex tasks
- Tools extend the agent's reach into the real world
- This simple formula creates powerful problem-solving systems

YHK

Practical Implementation Approach

- Start with building a basic ReAct agent using LangChain
- Identify drawbacks and limitations of the basic approach
- Understand where LangGraph comes into the picture
- Learn what specific problems LangGraph solves
- Progress from theory to practical coding implementation
- Build understanding through hands-on experience

YHK

Practical Implementation: LangChain ReACT way

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv
from langchain.agents import initialize_agent
from langchain_community.tools import TavilySearchResults

load_dotenv()

llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")

search_tool = TavilySearchResults(search_depth="basic")

tools = [search_tool]

agent = initialize_agent(tools=tools, llm=llm, agent="zero-shot-react-description", verbose=True)

agent.invoke("Give me a funny tweet about today's weather in Bangalore")
```

(Ref: LangGraph Crash Course - Harish Neel)

YHK

Practical Implementation: LangChain ReACT way

ReACT is nothing but a prompting technique.

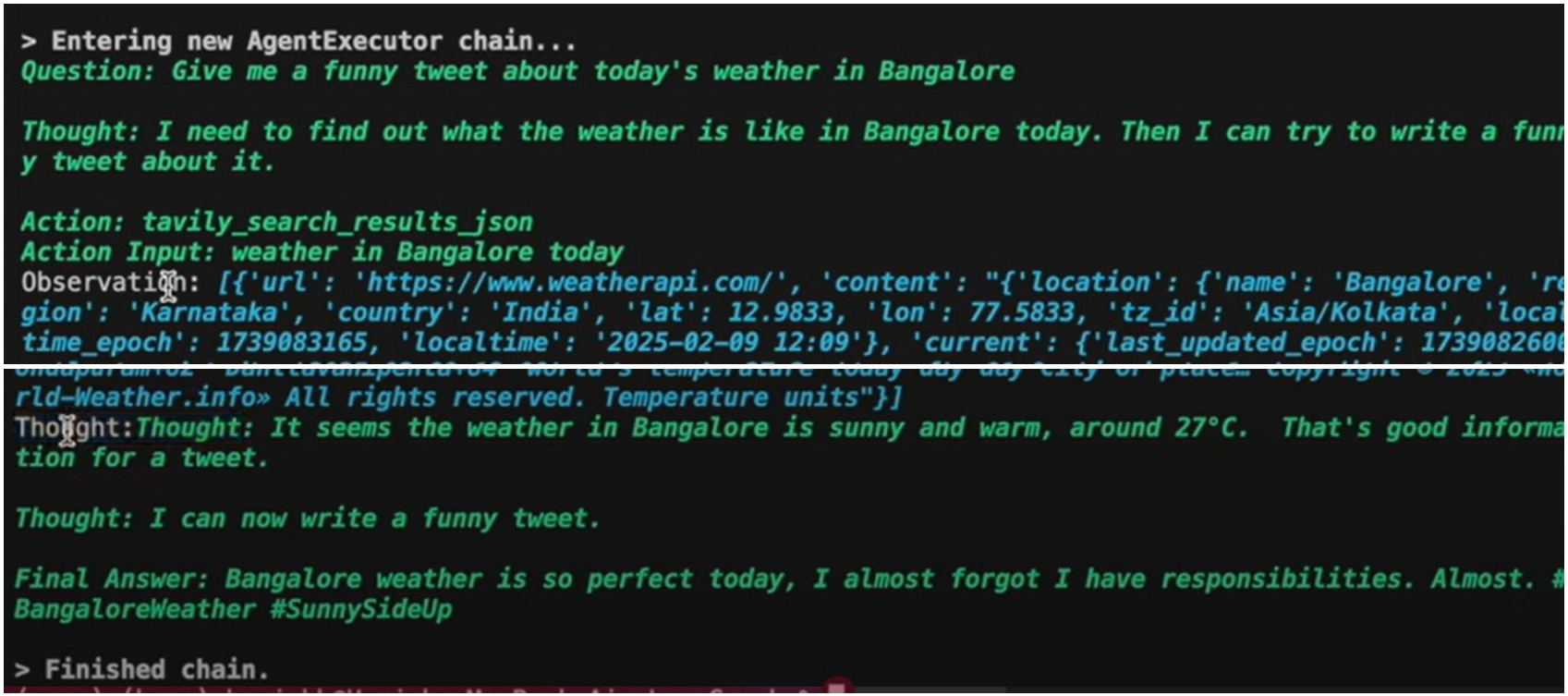
PromptTemplate

```
Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}
```

(Ref: LangGraph Crash Course - Harish Neel)

YHK

Study Output



(Ref: LangGraph Crash Course - Harish Neel)

YHK

Key Takeaways

- Agents represent autonomous AI that can make independent decisions
- ReAct pattern provides a structured framework for agent behavior
- Tools are essential for extending agent capabilities beyond reasoning
- The think-action-observe cycle mirrors human problem-solving
- LangChain orchestrates the interaction between reasoning and execution
- Understanding these fundamentals is crucial for advanced agent development

YHK

React Agents: Advantages and Flexibility

- React agents offer high flexibility - any state execution is possible
- Dynamic tool execution order: tool 1 $\rightarrow$ tool 2 or tool 2 $\rightarrow$ tool 1
- Example: SpaceX launch query could execute Tavily Search first, then get current time
- Alternative execution: get current system time first, then perform research
- Agents can adapt execution flow based on problem requirements
- Multiple valid execution paths for the same problem
- State transitions are not predetermined or fixed

YHK

React Agents: Drawbacks and Reliability Issues

- High flexibility often leads to reduced reliability
- Infinite loops are a significant problem with React agents
- Tools can get called repeatedly without proper termination
- Common causes of infinite loops:
  - Incorrectly defined tools
  - Insufficient LLM capabilities
  - Poor prompting without clear end conditions
- Execution is not completely under developer control
- Google Gemini example showed tool being called infinitely until stopped

YHK

Chains vs React Agents: The Trade-off

- **Chains**: Fixed assembly line approach
  - Low flexibility but high reliability
  - Predetermined execution sequence
  - Predictable outcomes
- **React Agents**: Dynamic execution approach
  - High flexibility but low reliability
  - Unpredictable execution paths
  - Potential for infinite loops
- Need for a solution combining best of both worlds
- Requirement: flexible yet reliable system

YHK

Introducing LangGraph: Best of Both Worlds

- LangGraph provides flexibility with reliability
- Uses state machine architecture for controlled execution
- Bridges the gap between rigid chains and unpredictable React agents
- Maintains dynamic execution while ensuring controllability
- Framework designed for building controllable agent workflows
- Combines the advantages while minimizing the drawbacks
- Graph-based approach enables structured flexibility

YHK

LangGraph: Definition and Core Purpose

- **Definition**: Framework for building controllable, persistent agent workflows
- Built-in support for human interaction and streaming
- Comprehensive state management capabilities
- Uses graph data structure as foundation
- Basic structure: Start Node → Agent → Action → Continue/End
- Enables creation of sophisticated agent workflows
- Designed for production-ready AI applications

YHK

LangGraph Key Features: Looping and Branching

- **Looping and Branching Abilities**:
  - Supports conditional statements and loop structures
  - Dynamic execution paths based on current state
  - Enables complex decision-making workflows
- **State Persistence**:
  - Automatic state saving and management
  - Support for pause and resume functionality
  - Ideal for long-running conversations and processes
- Provides controlled flexibility unlike pure React agents

YHK

LangGraph Key Features: Interaction and Integration

- **Human-Machine Interaction Support**:
  - Insert human review during execution
  - State editing and modification capabilities
  - Flexible interaction control mechanisms
- **Streaming Processing**:
  - Real-time feedback on execution status
  - Enhanced user experience through streaming output
- **Seamless LangChain Integration**:
  - Reuses existing LangChain components
  - Supports LangChain Expression Language
  - Rich tool and model support

YHK

Why Graph Data Structure?

- Graph structure chosen based on research paper evidence
- Most complex problem-solving research uses graph data structures
- Provides flexibility while maintaining controllability
- Enables autonomous decision-making capabilities
- Natural fit for agent workflow representation
- Balances flexibility with structured control
- Supports complex branching and conditional logic
- Research-proven approach for difficult AI problems

YHK

LangGraph Core Components Overview

- **Four Core Components**:
  - Nodes
  - Edges
  - Conditional Edges
  - State
- Components work together to create controllable workflows
- Each component serves a specific purpose in graph execution
- Understanding these components is crucial for LangGraph mastery
- Will be demonstrated through Reflection Agent Pattern example

YHK

Reflection Agent Pattern: Example Overview

- **Pattern Components**:
  - Generation Component: Creates content (e.g., tweets)
  - Criticism Component: Evaluates and suggests improvements
- **Workflow**:
  - Agent generates initial tweet
  - Critic evaluates quality and provides feedback
  - Generator improves tweet based on criticism
  - Process loops until quality threshold is met
- Example of iterative improvement through agent collaboration
- Demonstrates practical application of LangGraph components

YHK

Core Components: Nodes

- **Nodes**: Fundamental execution units in the graph
- Examples in Reflection Agent Pattern:
  - Start Node: Entry point of the workflow
  - Generate Tweet LLM: Content creation node
  - Criticize Tweet LLM: Evaluation node
  - End Node: Termination point
- Each node represents a specific operation or LLM call
- Nodes contain the actual logic and processing
- Can be simple operations or complex LLM interactions

YHK

Core Components: Edges

- **Edges**: Connections between nodes in the graph
- Represent the flow of execution from one node to another
- Shown as lines connecting different nodes
- Define the possible paths through the workflow
- Ensure proper sequence of operations
- Can be simple directional connections
- Enable linear workflow progression
- Essential for maintaining execution order

YHK

Core Components: Conditional Edges

- **Conditional Edges**: Decision points in the workflow
- Enable branching based on specific conditions
- In Reflection Agent example:
  - After tweet generation, can go to criticism OR end
  - Decision based on iteration count or quality threshold
- Represented by dotted lines in diagrams
- Allow for dynamic execution paths
- Essential for implementing complex decision logic
- Enable loops and conditional branching

YHK

Core Components: State Management

- **State**: Maintains context throughout workflow execution
- Preserves information between node executions
- In Reflection Agent Pattern:
  - Stores generated tweet content
  - Maintains criticism feedback
  - Tracks iteration count
  - Preserves conversation context
- Enables nodes to access and modify shared information
- Critical for maintaining workflow coherence
- Supports complex, stateful operations

YHK

Summary and Next Steps

- **Key Learnings**:
  - React agents: flexible but unreliable
  - LangGraph: combines flexibility with reliability
  - Graph structure enables controlled autonomous decision-making
  - Four core components: Nodes, Edges, Conditional Edges, State
- **Upcoming Topics**:
  - Deep dive into Reflection Agent Pattern
  - Hands-on implementation and coding
  - Real-world applications and use cases

YHK

Graph Data Structures

- Understanding graph-based architectures
- Directed Acyclic Graphs (DAGs) vs Cyclical Graphs
- Graph theory applications in AI agents
- Data flow management in graph structures
- Node relationships and connections
- State management across graph nodes
- Prerequisites for LangGraph implementation

YHK

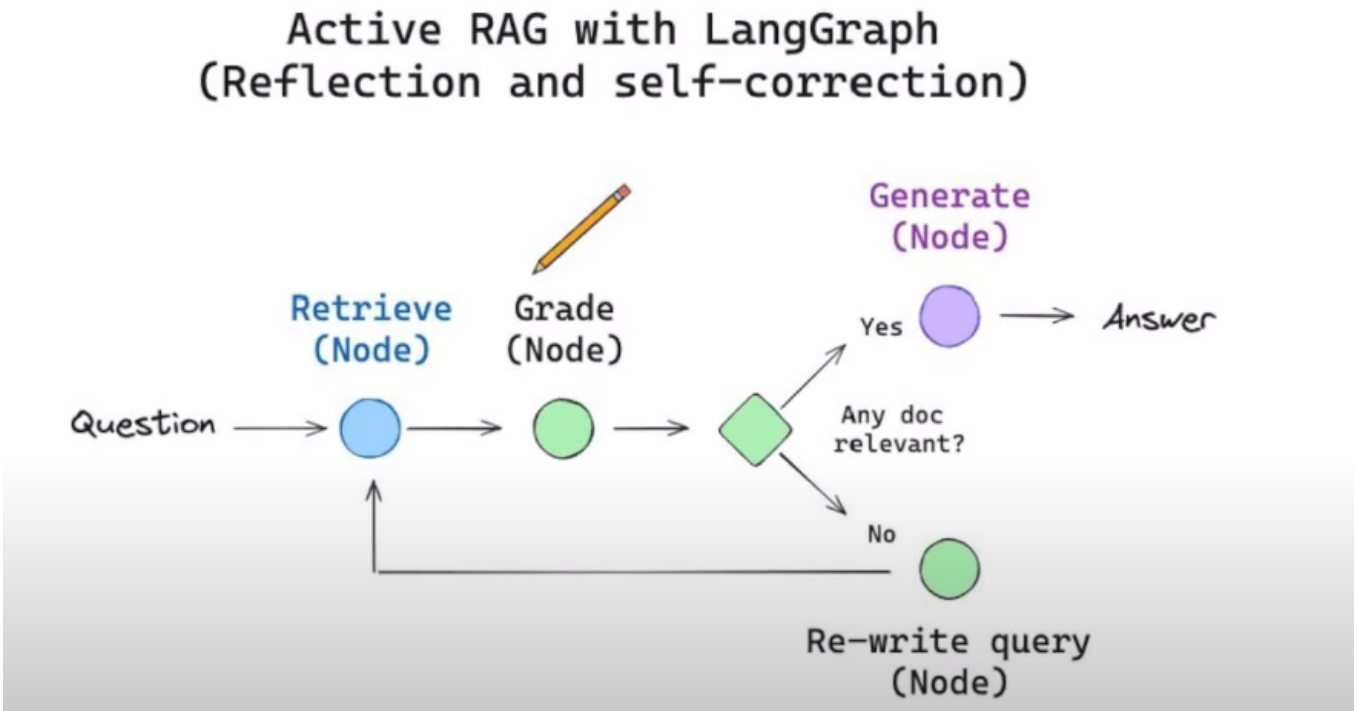
What is LangGraph and Why Use It?

- LangGraph: Advanced framework for building AI agent workflows
- Limitations of traditional LangChain approaches
- Enhanced flexibility and control over agent behavior
- Support for complex multi-step reasoning processes
- Better state management and workflow orchestration
- Low-level control for custom implementations
- Production-ready agent development capabilities

YHK

What is LangGraph?

- Building state-ful multi agent applications using LLMs (Large Language Models)
- So, components are not in 'chain' as in LangChain (ie Sequential) but can a Graph.
- Like Workflow modeling or orchestration



(Ref: Introduction to LangGraph — Building an AI Generated Podcast - Prompt Circle AI)

Introduction to LangGraph

- LangGraph is a graph-based framework for building complex LLM applications
- Represents AI workflows as directed graphs with nodes, edges, and data states
- Addresses limitations of LCEL and AgentExecutor in complex scenarios
- Provides intuitive and flexible approach to constructing conversational flows
- Supports looping, branching, and dynamic execution paths
- Enables state persistence and human-machine interaction
- Seamlessly integrates with existing LangChain ecosystem
- Offers streaming processing and real-time feedback capabilities

YHK

Limitations of LCEL Chain Expressions

- Linear processes with predefined execution order
- Difficulty in dynamic routing and conditional branching
- Complex state management in multi-turn conversations
- Manual state passing and updates increase code complexity
- Non-intuitive tool integration and coordination
- Nested tools create highly complex LCEL expressions
- Limited flexibility for dynamic conversational scenarios
- Error-prone state management across chain calls

YHK

LCEL Complexity Example

For example, when constructing a parallel sub-problem optimization chain for a problem decomposition strategy, the complexity of LCEL expressions becomes apparent. This example shows that when the nesting level of tools is slightly deeper, constructing LCEL chains becomes quite complex.

```
# Decomposition chain
decomposition_chain = (
    {"question": RunnablePassthrough()}
    | decomposition_prompt
    | ChatOpenAI(model="gpt-4o-mini", temperature=0)
    | StrOutputParser()
    | (lambda x: x.strip().split("\n"))
)
# Sub-question answer generation chain
sub_question_chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | sub_question_prompt
    | ChatOpenAI(model="gpt-4o-mini")
    | StrOutputParser()
)
# Assembly chain
chain = (
    {"question": RunnablePassthrough(), "context": decomposition_chain}
    | {"questions": RunnablePassthrough(), "answers": sub_question_chain.map()}
    | RunnableLambda(format_qa_pairs)
    | prompt
    | llm_output_str
)
```

YHK

Limitations of AgentExecutor

- Complex configuration for conversational flows and multi-turn dialogues
- Limited dynamic routing capability for conditional branches
- Lack of built-in state persistence mechanism
- Over-encapsulation with fixed input requirements
- Black-box uncontrollability in tool usage order
- Cannot insert human interaction during execution
- Difficult secondary development due to rigid structure
- Must restart from scratch when conversation resumes

YHK

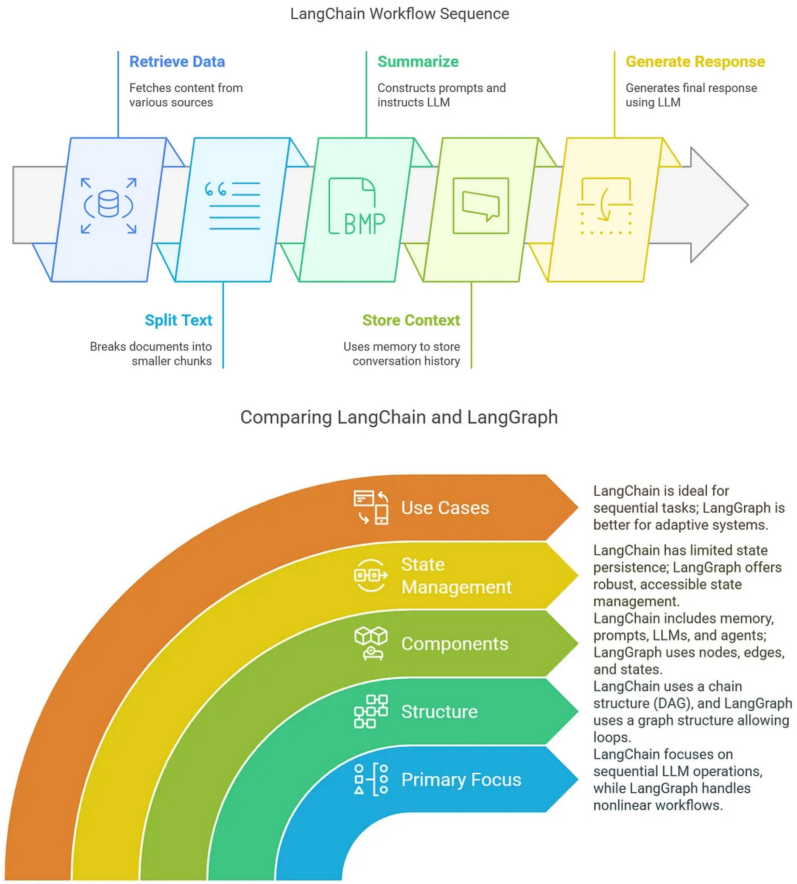
Why LangGraph?

- Limitations of LCEL and AgentExecutor, need a more flexible and powerful framework to build complex agent applications.
- Most of the Multi-agent frameworks are rigid, monolithic
- LangGraph decouples the agents from their orchestration.
- Here, agents can have different LLMs, can combine even non-LLM services together, can combine agents from other systems such as Autogen or Crew AI etc.

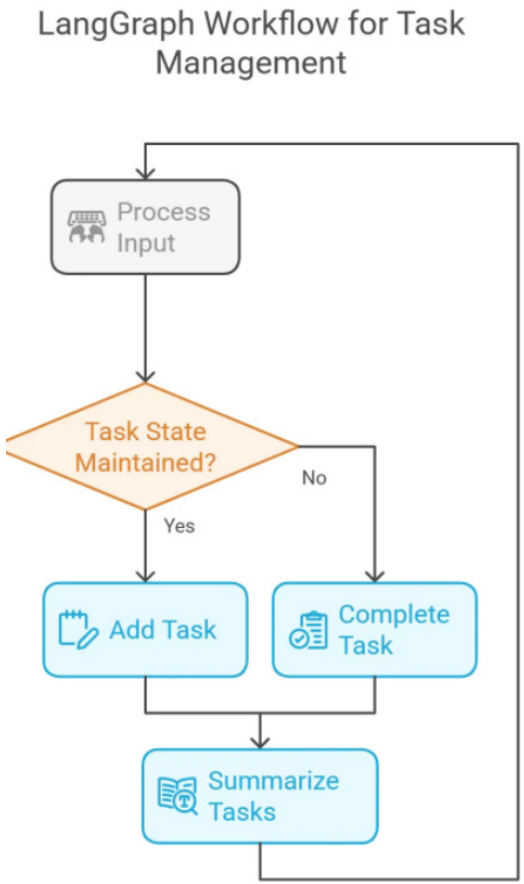
(Ref: Langgraph: The Agent Orchestrator - Rajib Deb)

YHK

LangChain vs LangGraph



(Ref: Vizuara AI Agents Bootcamp)



(Ref: Vizuara AI Agents Bootcamp)

LangChain vs LangGraph – Why a New Approach?

- LangChain excels at straightforward pipelines with fixed sequential tasks (retrieve → summarize → answer)
- Real-world workflows aren't always linear - they need branches, loops, and dynamic decision points
- LangGraph specializes in stateful, non-linear workflows within the LangChain ecosystem
- LangGraph exposes control flow explicitly, allowing fine-grained control over agent behavior
- You define a graph of possible steps instead of a rigid chain structure
- Agents can revisit nodes or choose different paths dynamically based on conditions
- Use LangChain for simple sequences, LangGraph for conditional logic and extended context
- LangGraph enables complex multi-agent systems with evolving task flows

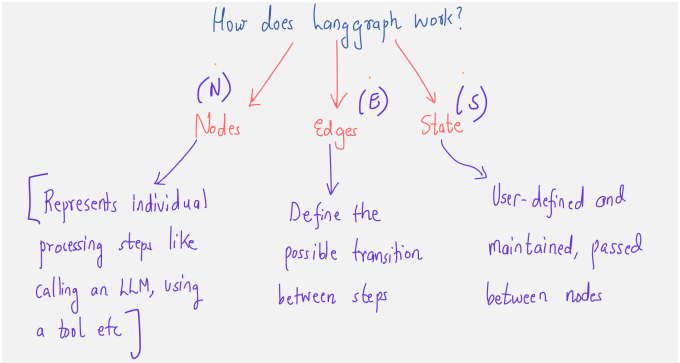
YHK

LangGraph Fundamentals: Nodes, Edges & State

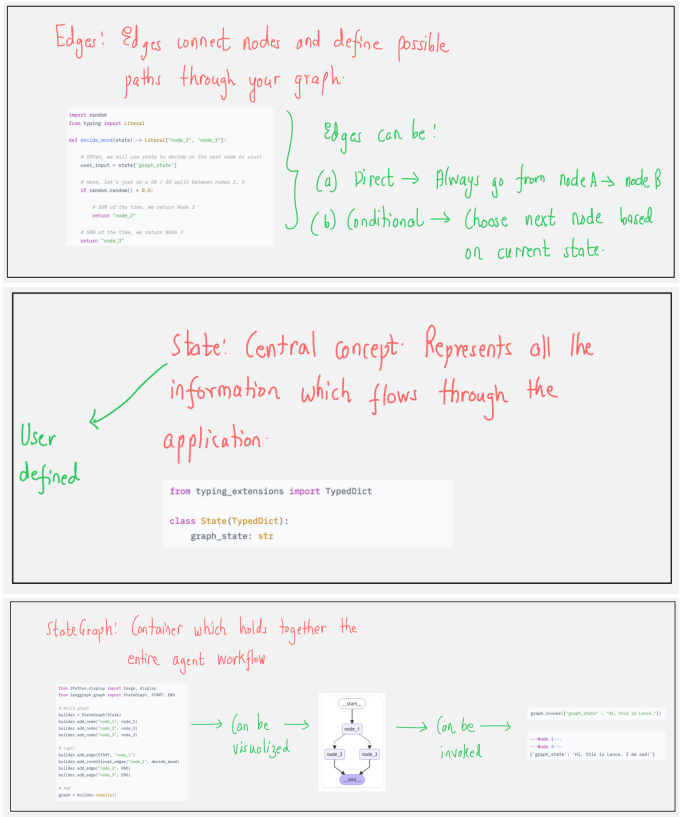
- **Nodes (N):** Individual processing steps implemented as Python functions that transform state
- Each node encapsulates one sub-task: LLM calls, calculations, file operations, or tool invocations
- **Edges (E):** Directed connections determining execution order and flow between nodes
- Edges can be linear progressions or conditional routes based on current state
- **State (S):** Shared data object (dictionary/TypedDict) persisting throughout agent execution
- State enables context and memory across workflow, allowing nodes to read previous results
- StateGraph ties everything together with designated START and END nodes
- This approach handles interactive, conditional loops that static chains struggle with

YHK

LangGraph



(Ref: Vizuara AI Agents Bootcamp)



(Ref: Vizuara AI Agents Bootcamp)

Agentic Architecture Patterns

- Reflection Agent: Self-evaluation and improvement mechanisms
- Reflexion Agent: Learning from past experiences
- Multi-agent Workflows: Collaborative agent systems
- Human-in-the-Loop Patterns: Manual oversight integration
- React Patterns: Reasoning and acting frameworks
- Chain Exploration: Alternative path discovery
- Pattern Selection: Choosing appropriate architectures

YHK

Key LangGraph Terminologies

- Graph: Overall workflow structure and organization
- State: Data persistence and management across nodes
- Node: Individual processing units within the graph
- Visualization: Graphical representation of workflows
- Breakpoints: Debugging and control mechanisms
- Edges: Connections and transitions between nodes
- Flow Control: Managing execution paths and decisions

YHK

Building LangGraph Chatbot

- Web search capabilities for real-time information
- Complex query routing to human reviewers
- Backward chaining for alternative exploration
- Human-in-the-loop integration patterns
- React pattern implementation for reasoning
- Multi-path exploration and decision making
- Production-ready chatbot architecture

YHK

Multi-Agent Systems

- Multiple agent creation and management
- Inter-agent communication protocols
- Task distribution and coordination
- More powerful than CrewAI with greater flexibility
- Low-level control for custom workflows
- Collaborative problem-solving approaches
- Scalable multi-agent architectures

YHK

RAG Integration with LangGraph

- Corrective RAG (C-RAG): Error correction mechanisms
- Adaptive RAG (A-RAG): Dynamic retrieval strategies
- Self-RAG: Self-evaluation and improvement
- Advanced retrieval and generation patterns
- Context-aware information processing
- Quality control in RAG implementations
- Performance optimization techniques

YHK

Persistence and Production Tools

- State persistence mechanisms in LangGraph
- LangGraph Studio: Development and visualization environment
- LangGraph Cloud API: Production deployment solutions
- Scalability considerations for production systems
- Monitoring and debugging capabilities
- Performance optimization strategies
- Enterprise-grade deployment patterns

YHK

Agents in Production

- Real-world use case exploration and implementation
- Production deployment best practices
- Future technology trends and opportunities
- Industry applications and case studies
- Scalability and performance considerations
- Monitoring and maintenance strategies
- Complete picture of AI agent technology future

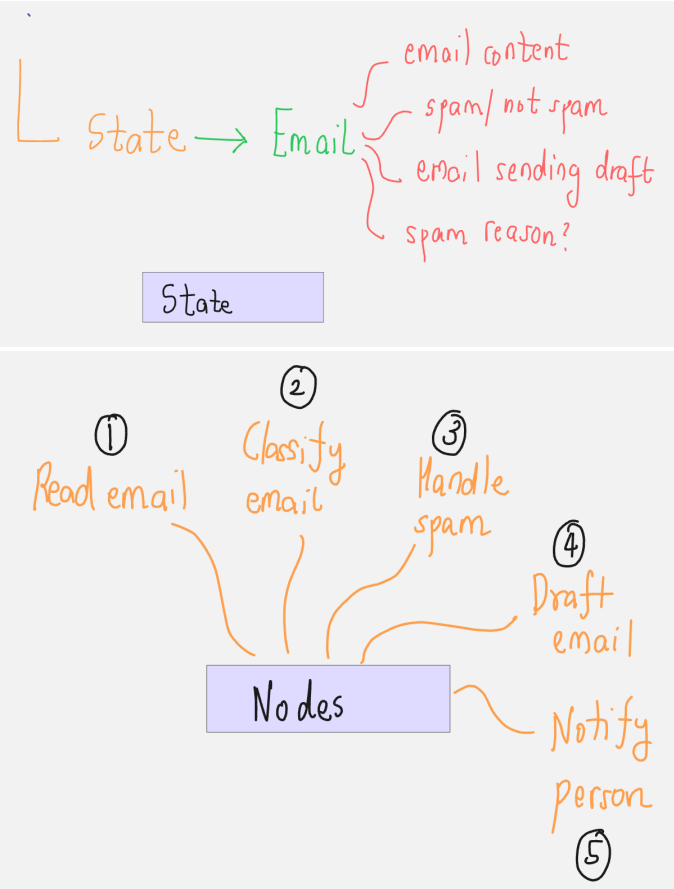
YHK

Building an Email Sorting Agent

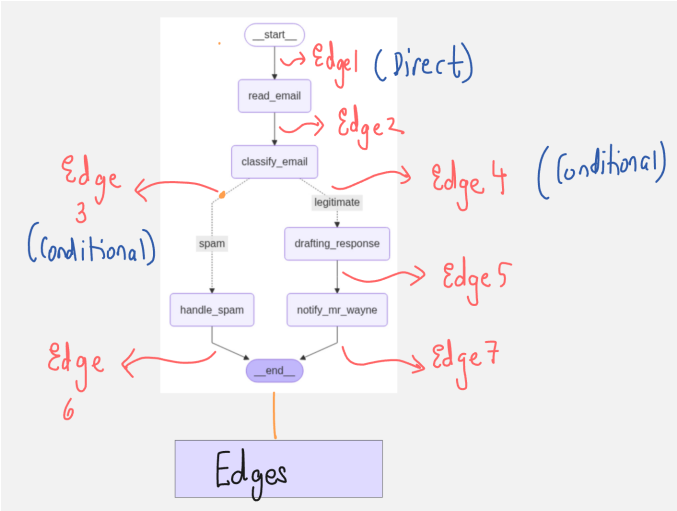
- Email processing assistant inspired by Alfred managing Bruce Wayne's inbox
- **EmailState:** Contains email content, spam flag, category, draft response, and message history
- **Five nodes:** read_email → classify_email → handle_spam/drafting_response → notify_mr_wayne
- Classify_email uses LLM to analyze content and set spam status in state
- Conditional edge routes to spam handler or response drafter based on classification
- Handle_spam terminates workflow early, drafting_response continues to notification
- Demonstrates clear if/else logic through graph structure rather than buried code
- Successfully tested on legitimate and crypto spam emails with correct routing

YHK

LangGraph



(Ref: Vizuara AI Agents Bootcamp)



(Ref: Vizuara AI Agents Bootcamp)

YHK

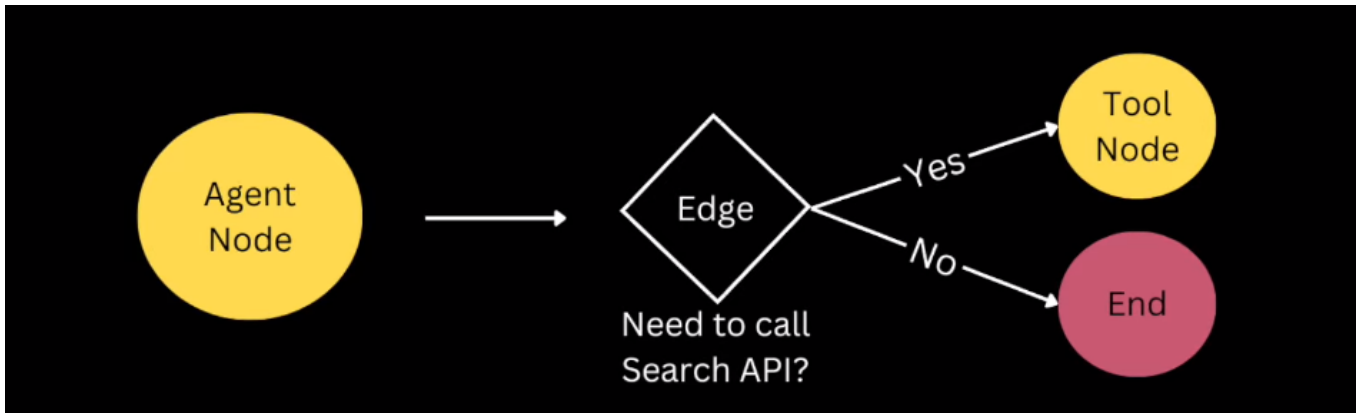
LangGraph

```
Processing legitimate email...
An AI Agent is processing an email from Cognizant AI with subject: Potential partnership !
ham

================================================
Hello, I am your AI Agent. You have received an email from Cognizant AI. I have already screened that it's not a spam.
Subject: Potential partnership !

I've prepared a draft response for your review:
------------------------------------------------
Subject: Re: Potential Partnership

Dear [Name],

Thank you for reaching out to us at Vizuara. We are excited about the possibility of collaborating with Cognizant AI and exploring potential partnership opportunities.

Dr. Raj would be pleased to discuss this further. Could you please suggest a few convenient times for a call or meeting? We will do our best to accommodate your schedule.

Looking forward to your response.

Best regards,

[Your Name]
[Your Position]
Vizuara
================================================


Processing spam email...
An AI Agent is processing an email from Crypto bro with subject: The best investment of 2025
spam
Alfred has marked the email as spam.
The email has been moved to the spam folder.
```

(Ref: Vizuara AI Agents Bootcamp)

Concepts

- Model: Large Language Model that supports Function Calling
- Tools: Actions taken by app, ie API calls, Db operations, etc
- State: Represents info that is carried though out the workflow e.g Message State has list of messages produced from each Node.
- Node: executable logic container, a Langchain runnable or a Tool invoker. Nodes are connected by edges.
- Edge: control flow of info, conditional or normal
- Workflow: The graph, having nodes and edges, can be invoked or streamed from.



(Ref: Introduction to LangGraph — Building an AI Generated Podcast - Prompt Circle AI)

Graph and State Machine Concepts

- State: System's behavioral condition (hunger, sleep, temperature)
- Events: Actions that trigger state changes (feeding, soothing)
- Nodes: Decision-makers and executors in the workflow
- Baby care analogy: Mother decides, elders check, father executes
- State updates trigger continuous decision-making cycles
- Graph structure provides flexible workflow representation
- State machines enable dynamic routing based on conditions
- More intuitive than linear processing chains

YHK

LangGraph Core Features

- Looping and branching capabilities with conditional statements
- State persistence with automatic save and restore functionality
- Human-machine interaction support with review and editing
- Streaming processing for real-time feedback and status updates
- Seamless integration with existing LangChain components
- Support for LCEL expressions and rich tool ecosystem
- Flexible interaction control mechanisms
- Enhanced user experience through responsive design

YHK

Core Concept: State

- State is the foundation of LangGraph applications
- Can be simple dictionary or Pydantic model
- Contains all information needed during runtime
- Passed between nodes and updated throughout execution

```python
from typing import List, Dict
from pydantic import BaseModel

class ChatState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    tools_output: Dict[str, str] = {}
    final_response: str = ""

```

YHK

Core Concept: Nodes

Nodes are Python functions used to process state and return updated state:

```python
async def process_input(state: ChatState) -> ChatState:
    # Process user input
    messages = state.messages + [{"role": "user", "content": state.current_input}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        tools_output=state.tools_output)

async def generate_response(state: ChatState) -> ChatState:
    # Generate response using LLM
    response = await llm.ainvoke(state.messages)
    messages = state.messages + [{"role": "assistant", "content": response}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        tools_output=state.tools_output,
        final_response=response)

```

YHK

Core Concept: Edges

- Edges define connections and routing logic between nodes
- Support conditional routing based on state
- Enable dynamic execution paths

```
from langgraph.graph import StateGraph, END

# Create graph structure
workflow = StateGraph(ChatState)

# Add nodes
workflow.add_node("process_input", process_input)
workflow.add_node("generate_response", generate_response)

# Define edges and routing logic
workflow.add_edge("process_input", "generate_response")
workflow.add_edge("generate_response", END)

```

YHK

Building the Chatbot Graph

This example demonstrates the basic usage of LangGraph:

- Define state model
- Create processing nodes
- Build the graph structure
- Define routing logic
- Compile and run

YHK

Simple Chatbot Implementation

```python
from typing import List, Dict
from pydantic import BaseModel
from langgraph.graph import StateGraph, END
from langchain_core.language_models import ChatOpenAI

class ChatState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    should_continue: bool = True

async def process_user_input(state: ChatState) -> ChatState:
    messages = state.messages + [{"role": "user", "content": state.current_input}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        should_continue=True
    )

async def generate_ai_response(state: ChatState) -> ChatState:
    llm = ChatOpenAI(temperature=0.7)
    response = await llm.ainvoke(state.messages)
    messages = state.messages + [{"role": "assistant", "content": response}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        should_continue=True
    )

def should_continue(state: ChatState) -> str:
    if "goodbye" in state.current_input.lower():
        return "end"
    return "continue"
```

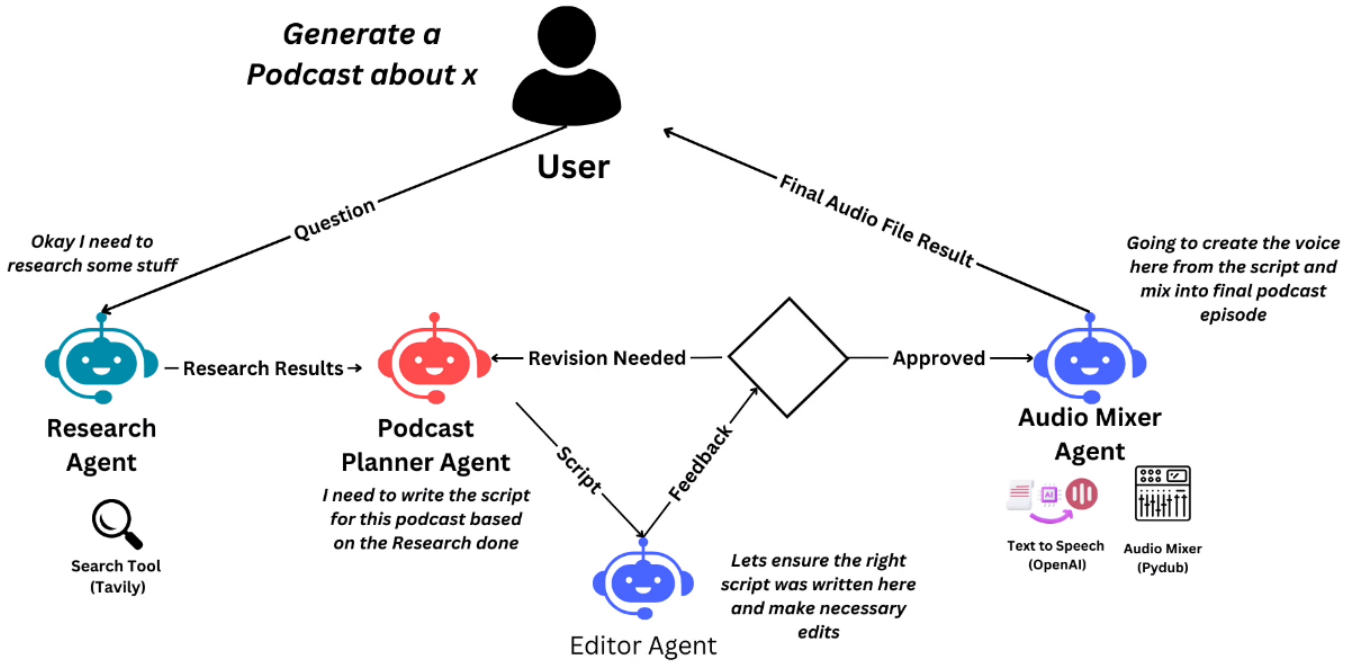
Building the Chatbot Graph

```
# Build the graph
workflow = StateGraph(ChatState)

# Add nodes
workflow.add_node("process_input", process_user_input)
workflow.add_node("generate_response", generate_ai_response)

# Add edges
workflow.add_edge("process_input", "generate_response")
workflow.add_conditional_edges("generate_response", should_continue,
    {"continue": "process_input", "end": END})

# Compile the graph
app = workflow.compile()

# Run the conversation
async def chat():
    state = ChatState()
    while True:
        user_input = input("You: ")
        state.current_input = user_input
        state = await app.ainvoke(state)
        print("Bot:", state.messages[-1]["content"])
        if not state.should_continue:
            break

# Run chat
import asyncio
asyncio.run(chat())
```

YHK

Applications

Podcast Generator

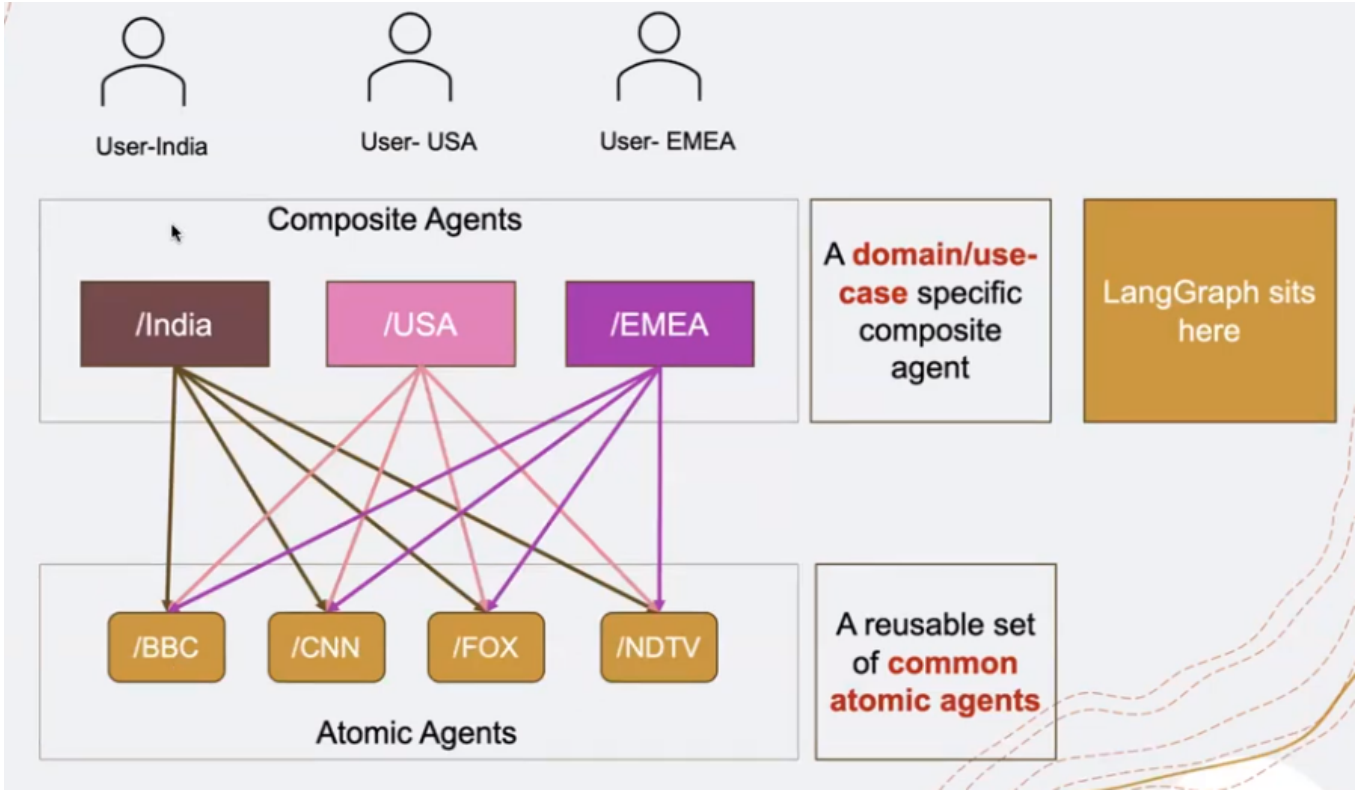


(Ref: Introduction to LangGraph — Building an AI Generated Podcast - Prompt Circle AI)

Code at https://github.com/hollaugo/langgraph-framework-tutorial

YHK

News Aggregator



Code:
https://github.com/rajib76/multi_agent/blob/main/01_how_to_langgraph_example_01.p

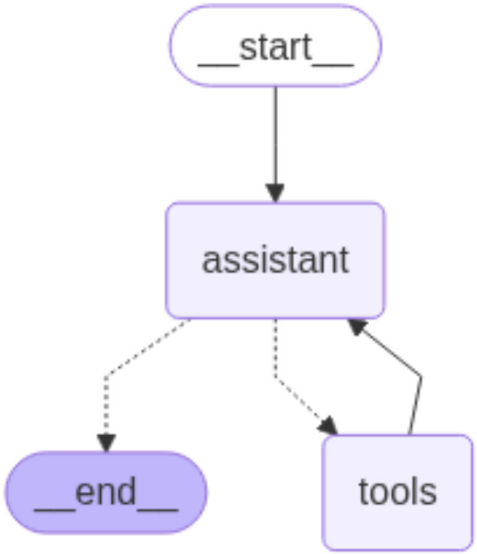
(Ref: Langgraph: The Agent Orchestrator - Rajib Deb)

YHK

Agents using LangGraph

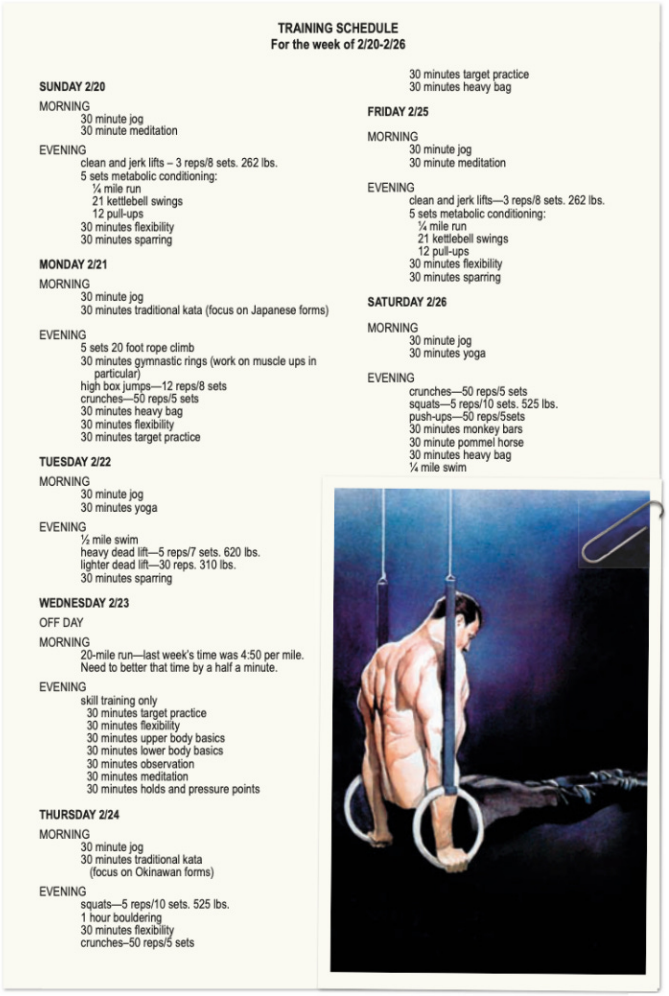
Creating a Vision Assistant Agent

- Vision-enabled agent implementing explicit "Thought → Action → Observation" loop
- **AgentState:** Stores input_file (image path) and messages list for conversation history
- **Assistant node:** GPT-4 with vision capability bound to image analysis and math tools
- **Tools node:** Executes tool functions (extract_text, divide) and records results
- Assistant decides to use tools or provide final answer based on task requirements
- Conditional edge uses tools_condition to route between assistant and tools nodes
- Creates cycle: assistant thinks → tool execution → assistant processes results → repeat
- Loop continues until assistant provides final answer without requesting tools



YHK

Creating a Vision Assistant Agent



TRAINING SCHEDULE
For the week of 2/20-2/26

SUNDAY 2/20

MORNING
30 minute jog
30 minute meditation

EVENING
clean and jerk lifts – 3 reps/8 sets. 262 lbs.
5 sets metabolic conditioning:
¼ mile run
21 kettlebell swings
12 pull-ups
30 minutes flexibility
30 minutes sparring

MONDAY 2/21

MORNING
30 minute jog
30 minutes traditional kata (focus on Japanese forms)

EVENING
5 sets 20 foot rope climb
30 minutes gymnastic rings (work on muscle ups in particular)
high box jumps—12 reps/8 sets
crunches—50 reps/5 sets
30 minutes heavy bag
30 minutes flexibility
30 minutes target practice

TUESDAY 2/22

MORNING
30 minute jog
30 minutes yoga

EVENING
½ mile swim
heavy dead lift—5 reps/7 sets. 620 lbs.
lighter dead lift—30 reps. 310 lbs.
30 minutes sparring

WEDNESDAY 2/23

OFF DAY

MORNING
20-mile run—last week's time was 4:50 per mile.
Need to better that time by a half a minute.

EVENING
skill training only
30 minutes target practice
30 minutes flexibility
30 minutes upper body basics
30 minutes lower body basics
30 minutes observation
30 minutes meditation
30 minutes holds and pressure points

THURSDAY 2/24

MORNING
30 minute jog
30 minutes traditional kata
(focus on Okinawan forms)

EVENING
squats—5 reps/10 sets. 525 lbs.
1 hour bouldering
30 minutes flexibility
crunches–50 reps/5 sets
30 minutes target practice
30 minutes heavy bag

FRIDAY 2/25

MORNING
30 minute jog
30 minute meditation

EVENING
clean and jerk lifts—3 reps/8 sets. 262 lbs.
5 sets metabolic conditioning:
¼ mile run
21 kettlebell swings
12 pull-ups
30 minutes flexibility
30 minutes sparring

SATURDAY 2/26

MORNING
30 minute jog
30 minutes yoga

EVENING
crunches—50 reps/5 sets
squats—5 reps/10 sets. 525 lbs.
push-ups—50 reps/5sets
30 minutes monkey bars
30 minute pommel horse
30 minutes heavy bag
¼ mile swim

In an effort to inspire the all-important Dark Knight to take time out of his busy schedule and actually consume a reasonable amount of sustenance, I have taken the liberty of composing a menu for today's scheduled meals. It is my high hope that these elegantly prepared courses will not share the fate of their predecessors resting cold and untouched on a computer console.
– A

…NE MANOR

…esday's Menu

Breakfast

six poached eggs laid over artichoke bottoms with a sage pesto sauce
thinly sliced baked ham
mixed organic fresh fruit bowl
freshly squeezed orange juice
organic, grass-fed milk
4 grams branched-chain amino acid
2 grams fish oil

Lunch

local salmon with a ginger glaze
organic asparagus with lemon garlic dusting
Asian yam soup with diced onions
2 grams fish oil

Dinner

grass-fed local sirloin steak
bed of organic spinach and piquillo peppers
oven-baked golden herb potato
2 grams fish oil

(Ref: Vizuara AI Agents Bootcamp)

YHK

Creating a Vision Assistant Agent

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

According to the note provided by MR Wayne in the provided images, what's the list of items I should buy for the dinner menu?
================================== Ai Message ==================================
Tool Calls:
  extract_text (call_4zjSSLKfjPgjoYev9B0sBKD4)
 Call ID: call_4zjSSLKfjPgjoYev9B0sBKD4
  Args:
    img_path: Batman_training_and_meals.png
================================= Tool Message =================================
Name: extract_text

TRAINING SCHEDULE
For the week of 2/20-2/26

SUNDAY 2/20
MORNING
30 minute jog
30 minute meditation

EVENING
clean and jerk lifts-3 reps/8 sets. 262 lbs.
5 sets metabolic conditioning:
2 min kettlebell swings
2 min pull-ups
2 minutes flexibility
30 minutes sparring

MONDAY 2/21
MORNING
30 minute jog
30 minutes traditional kata (focus on Japanese forms)
```

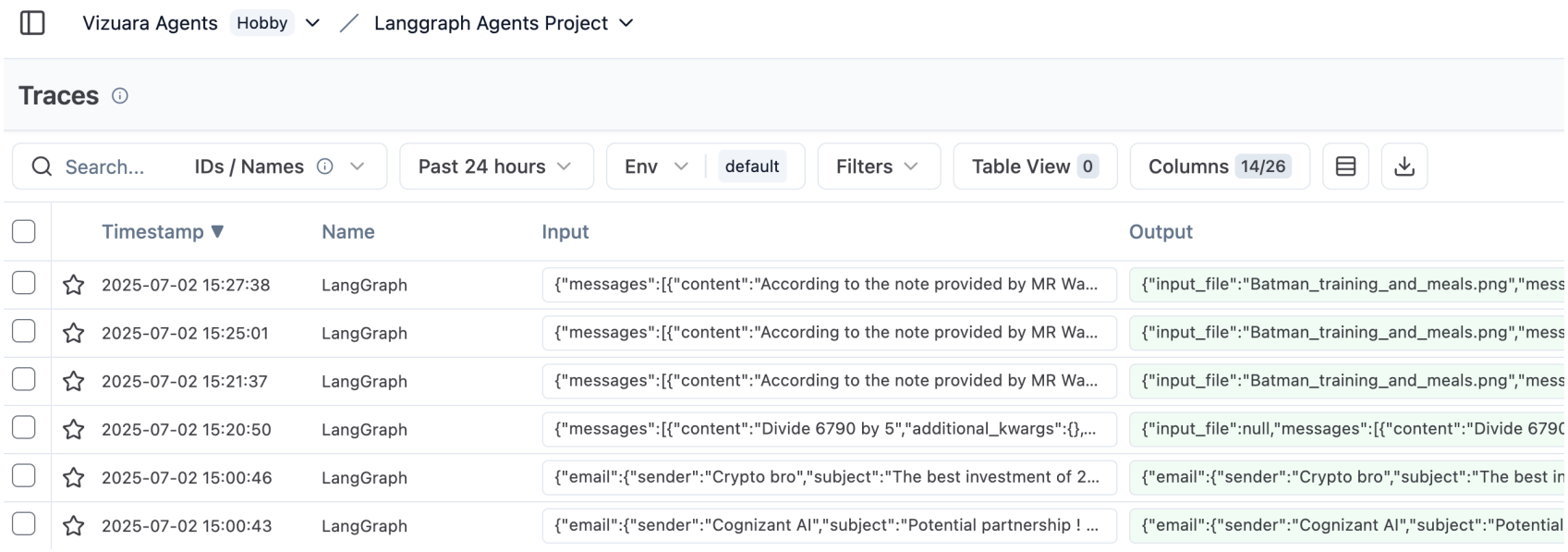
(Ref: Vizuara AI Agents Bootcamp)

Tracing Agent Workflows with Langfuse

- Observability crucial for debugging complex agent behaviors with multiple steps
- **Traces:** Detailed execution records capturing prompts, responses, paths, and tool usage
- Langfuse provides user-friendly dashboard for inspecting agent workflow traces
- Integrates via OpenTelemetry standards with tracing callbacks in LangGraph code
- Enables monitoring of both email agent and vision agent execution flows
- Tools-assistant loop clearly visible in Langfuse trace visualization
- Answers "why did my agent do X?" through accessible trace inspection
- Essential best practice for moving from prototypes to production-grade AI systems

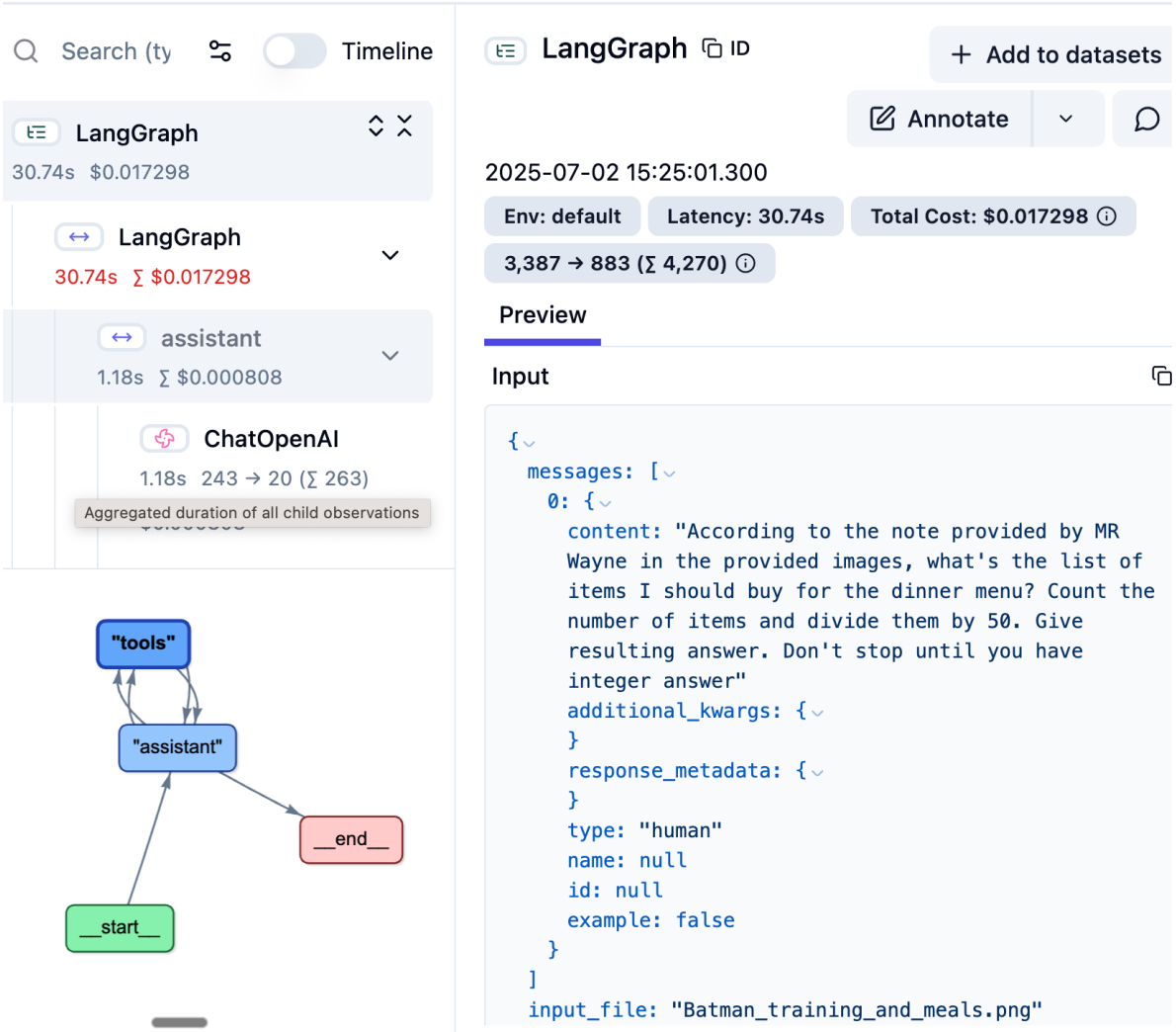
YHK

Tracing Agent Workflows with Langfuse



(Ref: Vizuara AI Agents Bootcamp)

Tracing Agent Workflows with Langfuse



(Ref: Vizuara AI Agents Bootcamp)

Best Practices

- State Design: Keep the state model simple and clear, only including necessary information. Use type hints to increase code readability.
- Node Functions: Maintain single responsibility, handle exceptions, and return new state objects instead of modifying existing state.
- Edge Design: Use clear conditional logic, avoid complex cyclic dependencies, and consider all possible paths.
- Error Handling: Add error handling at critical nodes, provide fallback mechanisms, and log detailed error information.

YHK

Conclusions

Best Practices and Considerations

- Keep state models simple and clear with necessary information only
- Maintain single responsibility in node functions
- Handle exceptions and return new state objects
- Use clear conditional logic in edge design
- Avoid complex cyclic dependencies in workflow
- Add error handling at critical nodes with fallback mechanisms
- Consider performance impacts of checkpoint mechanism
- Choose appropriate data processing methods for use cases

YHK

Future Developments and Limitations

- Current streaming has long waiting times for LLM nodes
- Ideal: Node-level streaming within graph streaming
- Market agents provide better step-by-step streaming
- LangGraph rapidly evolving with frequent updates
- Pre-built components may change in future versions
- Monitor documentation for latest features and changes
- Core design philosophy remains valuable for learning
- Expected improvements in streaming processing capabilities

YHK

Conclusion

- LangGraph addresses LCEL and AgentExecutor limitations effectively
- Graph-based approach provides intuitive workflow representation
- Advanced features support complex AI application development
- State management and persistence enable long-running conversations
- Human-in-the-loop capabilities enhance decision quality
- Modular subgraph architecture improves maintainability
- Streaming responses provide real-time user feedback
- Continuous evolution promises enhanced capabilities for AI development

YHK

References

Many publicly available resources have been refereed for making this presentation. Some of the notable ones are:

- LangGraph Crash Course - Harish Neel
- LangGraph Advanced Tutorial - James Li
- Learn LangGraph - The Easy Way, very nice explanation
- LangGraph Crash Course with code examples - Sam Witteveen
- Official Site: https://python.langchain.com/docs/langgraph https://github.com/langchain-ai/langgraph/tree/main
- LangGraph (Python) Series
- Introduction to LangGraph — Building an AI Generated Podcast
- Langgraph: The Agent Orchestrator - Rajib Deb
- LangGraph Deep Dive: Build Better Agents James Briggs

YHK

Thanks ...

- Search **"Yogesh Haribhau Kulkarni"** on Google and follow me on LinkedIn and Medium
- Office Hours: Saturdays, 2 to 3 pm (IST); Free-Open to all; email for appointment.
- Email: yogeshkulkarni at yahoo dot com

(https://www.linkedin.com/in/yogeshkulkarni/)

(https://medium.com/@yogeshharibhaukulkarni)

(https://www.github.com/yogeshhk/)

YHK