Zero-to-Hero: LangGraph

Yogesh Haribhau Kulkarni

YHK

Outline

About Me

YHK

Yogesh Haribhau Kulkarni

Bio:

- 20+ years in CAD/Engineering software development
- Got Bachelors, Masters and Doctoral degrees in Mechanical Engineering (specialization: Geometric Modeling Algorithms).
- Currently doing Coaching in fields such as Data Science, Artificial Intelligence Machine-Deep Learning (ML/DL) and Natural Language Processing (NLP).
- Feel free to follow me at:
  - Github (github.com/yogeshhk)
  - LinkedIn (www.linkedin.com/in/yogeshkulkarni/)
  - Medium (yogeshharibhaukulkarni.medium.com)
  - Send email to yogeshkulkarni at yahoo dot com

Office Hours: Saturdays, 2 to 5pm (IST); Free-Open to all; email for appointment.

YHK

Introduction to LangGraph

(Ref: LangGraph Crash Course - Harish Neel)

YHK

Background

The Evolution of LLM Applications

- **Early Days**: Simple prompt-response patterns with single LLM calls
- **Challenge**: Real-world problems require multiple steps, decisions, and tool usage
- **Question**: How do we give LLMs more capability while maintaining control?
- **Journey**: From deterministic code $\rightarrow$ intelligent chains $\rightarrow$ autonomous agents
- **Core Tension**: Freedom vs Reliability
  - More autonomy = More capable, but less predictable
  - More structure = More reliable, but less flexible
- **Goal**: Find the sweet spot between flexibility and control

YHK

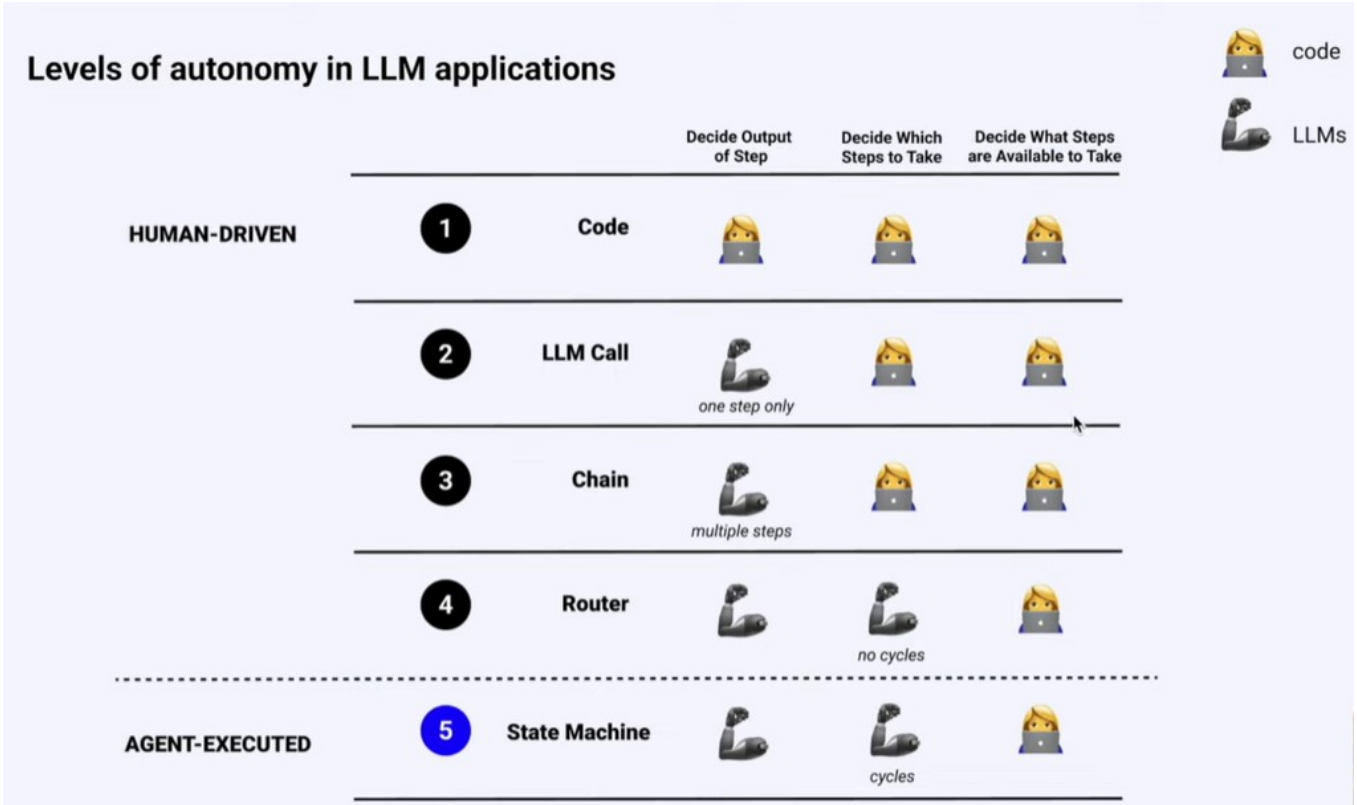
Comparison of Autonomy Levels

Level	Description	Limitation
Code	100% deterministic, rule-based logic.	Must anticipate every scenari
LLM Call	A single, atomic intelligent task like summarization.	Cannot break down complex step problems.
Chains	A fixed, unidirectional sequence of operations (A $\rightarrow$ B $\rightarrow$ C).	Inflexible; cannot adapt to d inputs or scenarios.
Routers	Chains with conditional branching, but still unidirectional.	No loops or cycles; cannot rev vious steps to correct mistake
State (LangGraph)	A state machine with loops and branches. The workflow is defined by the developer, but the path can be cyclic (A $\rightarrow$ B $\leftrightarrow$ C).	The flow is prescribed by the oper; not fully autonomous.
Agents	Fully autonomous decision-making. The agent independently decides the next step in a Think $\rightarrow$ Act $\rightarrow$ Observe	Can be unpredictable and is p looping infinitely without stri trols.

YHK

The Spectrum of LLM Application Autonomy

- LLM applications exist on a spectrum, balancing developer control with AI flexibility. The goal is to find the sweet spot for your use case.
- **Core Tension**: More control means more reliability but less flexibility. More autonomy means more capability but less predictability.
- The evolution moves from simple, deterministic code to complex, autonomous agents capable of cyclic reasoning and self-correction.



Human vs Agent Driven

- 1-4 Human driven vs 5-6 Agent driven
- Why '5' ie 'State' agent driven? and not '3' or '4' where LLM is playing a part.
- A chain or a router is unidirectional hence it is not an agent
- A state machine can go back in the flow as we have cycles and the flow is controlled by the llm hence it is called an agent.
- '3-4'has no self-correction, no training/learning, no refinement. That's intelligence. That's Agent.

YHK

Introduction to LangGraph

YHK

What is LangGraph?

- LangGraph is an extension of the popular LangChain library.
- Allows you to create AI applications that can perform multiple steps, make decisions, and maintain information across those steps.
- Think of it like building a flowchart for your AI to follow.

YHK

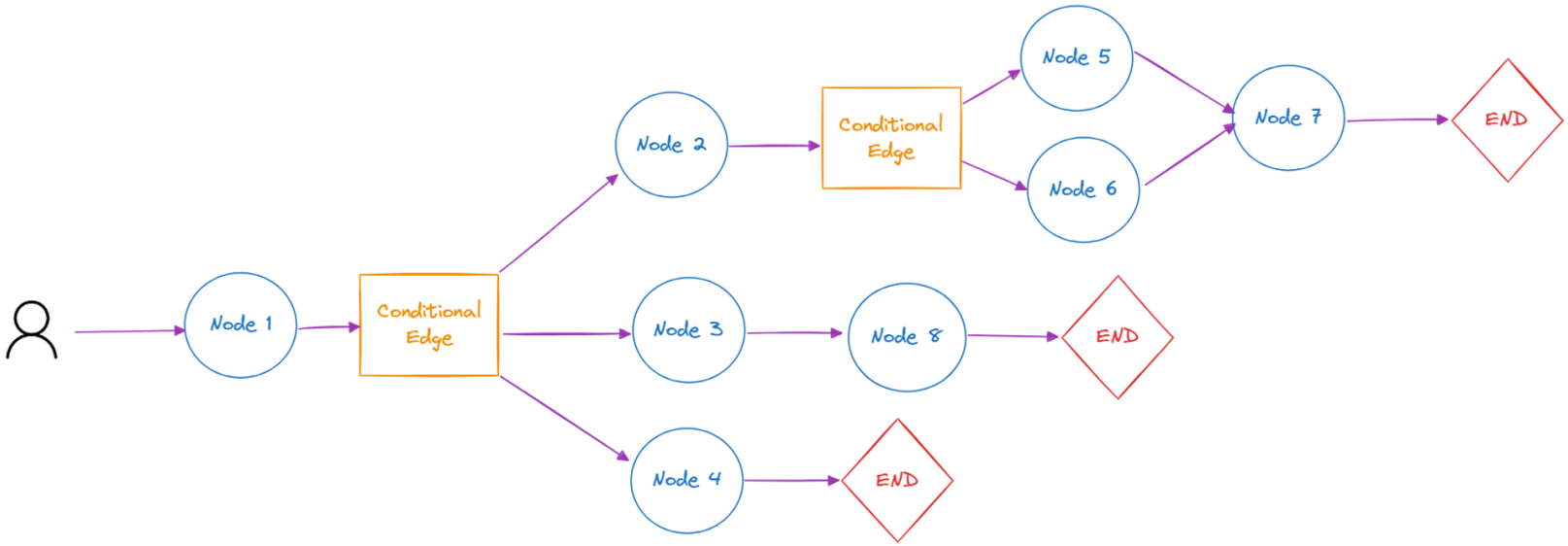
Why LangGraph? - The Comprehensive Pitch

- **Problem**: Traditional chains are too rigid, pure agents are too unpredictable
- **Solution**: LangGraph provides controlled flexibility through graph-based state machines
- **Key Benefits**:
  - Explicit control flow with loops, branches, and conditional logic
  - Built-in state persistence for long-running workflows
  - Human-in-the-loop capabilities at any point
  - Streaming support for real-time feedback
  - Production-ready with debugging and observability
- **Use Cases**: Multi-step reasoning, workflow automation, complex decision trees, collaborative agents
- **Result**: Reliable autonomous systems that combine flexibility with predictability

YHK

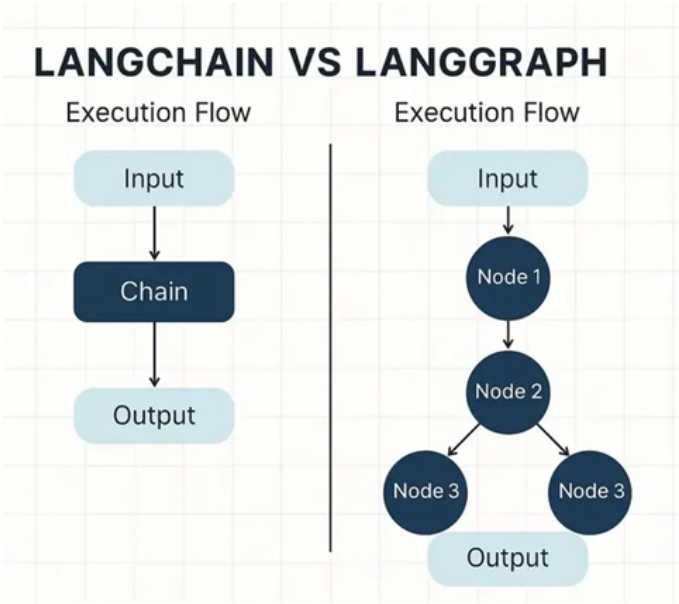
How is LangGraph Different?

- Gives you more control over how your AI makes decisions
- Allows your AI to revisit previous steps if needed
- Makes it easier to add human oversight at specific points
- Integrates smoothly with other LangChain tools



Agents in LangChain vs LangGraph

- **LangChain Agents (AgentExecutor)**:
  - Black-box execution with limited control
  - Fixed ReAct pattern implementation (TAO: Think-Action-Observe loop till we get the final answer)
  - Cannot pause, resume, or modify mid-execution
  - No state persistence across sessions
  - Prone to infinite loops without guardrails
- **LangGraph Agents**:
  - White-box with full visibility into execution
  - Custom patterns beyond ReAct
  - Pause, resume, edit state at any node
  - Persistent state with checkpointing
  - Explicit cycle limits and control flow
- **Bottom Line**: LangChain agents for simple tasks, LangGraph for production systems



Why Graph Structure Over Chains?

- **Chains Limitation**: Linear, unidirectional flow - A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ End
- **Real-World Problems**: Require branching, loops, and conditional paths
- **Graph Advantages**:
  - Conditional branching: "If quality ¡ threshold, loop back"
  - Multiple paths: "Route to specialized nodes based on task type"
  - Cycles: "Iterate until convergence or max iterations"
  - Parallel execution: "Process multiple sub-tasks simultaneously"
- **Research-Backed**: Most complex AI problem-solving papers use graph structures
- **Natural Fit**: Mirrors human problem-solving with think-act-observe-decide cycles
- Graphs provide flexibility while maintaining explicit control flow

YHK

Agentic Patterns: Autonomous vs Workflow Automation

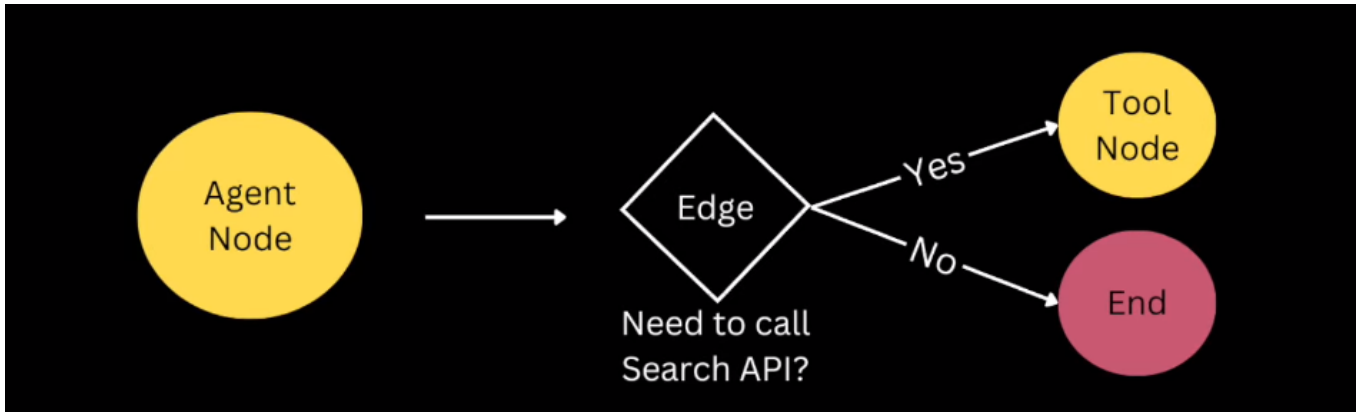
- **Autonomous Agents**:
  - High-level goals with minimal constraints
  - Self-directed exploration and decision-making
  - Example: "Research this topic and write a report"
  - Higher risk of unpredictability
- **Workflow Automation Agents**:
  - Structured processes with defined checkpoints
  - Predictable paths with conditional logic
  - Example: "Review email → Classify → Route → Draft response"
  - Balance between automation and control
- **LangGraph Sweet Spot**: Workflow automation with controlled autonomy
- Graphs enable explicit workflow definition while allowing intelligent decisions at each node

YHK

LangGraph Core Concepts

LangGraph Concepts

- Model: Large Language Model that supports Function Calling
- Tools: Actions taken by app - API calls, database operations, etc.
- State: Information carried throughout the workflow (e.g., Message State)
- Node: Executable logic container - a LangChain runnable or Tool invoker
- Edge: Control flow of information - conditional or normal
- Workflow: The graph with nodes and edges that can be invoked or streamed

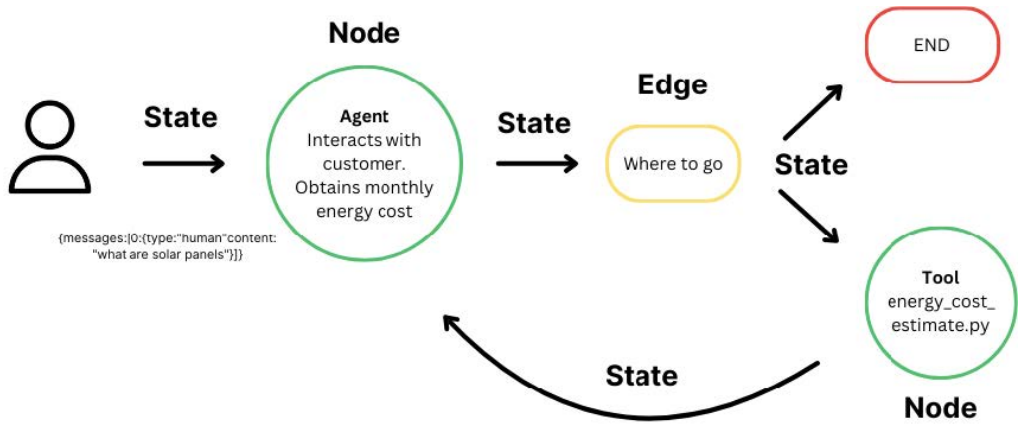


(Ref: Introduction to LangGraph — Building an AI Generated Podcast - Prompt Circle AI)

YHK

LangGraph Fundamentals: Nodes, Edges & State

- **Nodes (N):** Individual processing steps as Python functions that transform state
- Each node encapsulates one sub-task: LLM calls, calculations, tool invocations
- **Edges (E):** Directed connections determining execution flow between nodes
- Edges can be linear or conditional routes based on current state
- **State (S):** Shared data object persisting throughout execution

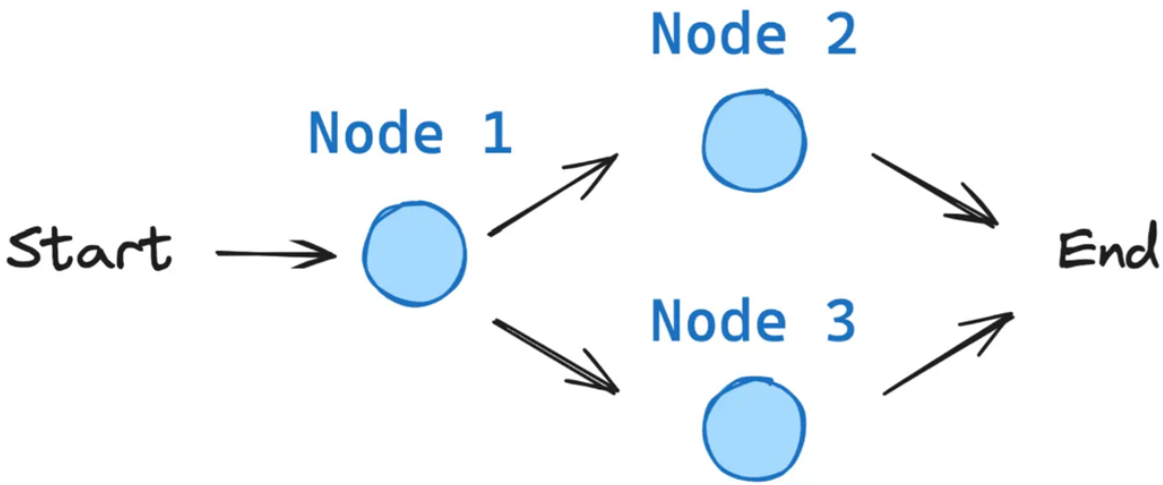


(Ref: The Complete Guide to Building LangChain Agents)

YHK

Core Components: Nodes

- Fundamental execution units in the graph
- Each node represents a specific operation or LLM call
- Nodes contain the actual logic and processing
- Can be simple operations or complex LLM interactions
- Examples: Start Node, Generate Content, Evaluate Quality, Tool Invocation, End Node



(Ref: Building Simple LangGraph - Uss Varma)

YHK

Is Each Node an Agent or Is the Graph an Agent?

- **The Graph is the Agent** - not individual nodes
- **Nodes are**: Individual processing units/functions
  - Can be simple Python functions
  - Can be LLM calls
  - Can be tool invocations
  - Can be sub-agents themselves
- **The Graph as Agent**:
  - The complete workflow represents the agent's behavior
  - State flows through nodes, creating agent "memory"
  - Control flow (edges) represents agent's decision-making
  - Overall graph exhibits autonomous, goal-directed behavior
- **Analogy**: Nodes are like neurons, the graph is the brain
- Individual nodes are stateless functions; the graph maintains state

YHK

What Can Nodes Do? - Part 1

- **Nodes are Extremely Flexible** - any Python function
- **1. Simple LLM Calls**:
  - Generate text, classify, extract information
  - Example: ‘llm.invoke("Summarize this text")‘
- **2. RAG Operations**:
  - Retrieve documents from vector database
  - Rerank results
  - Generate answers with retrieved context
  - Example: Vector search → Context → LLM generation
- **3. API Calls**:
  - External service calls (weather, database, CRM)
  - Example: Fetch customer data from Salesforce API

YHK

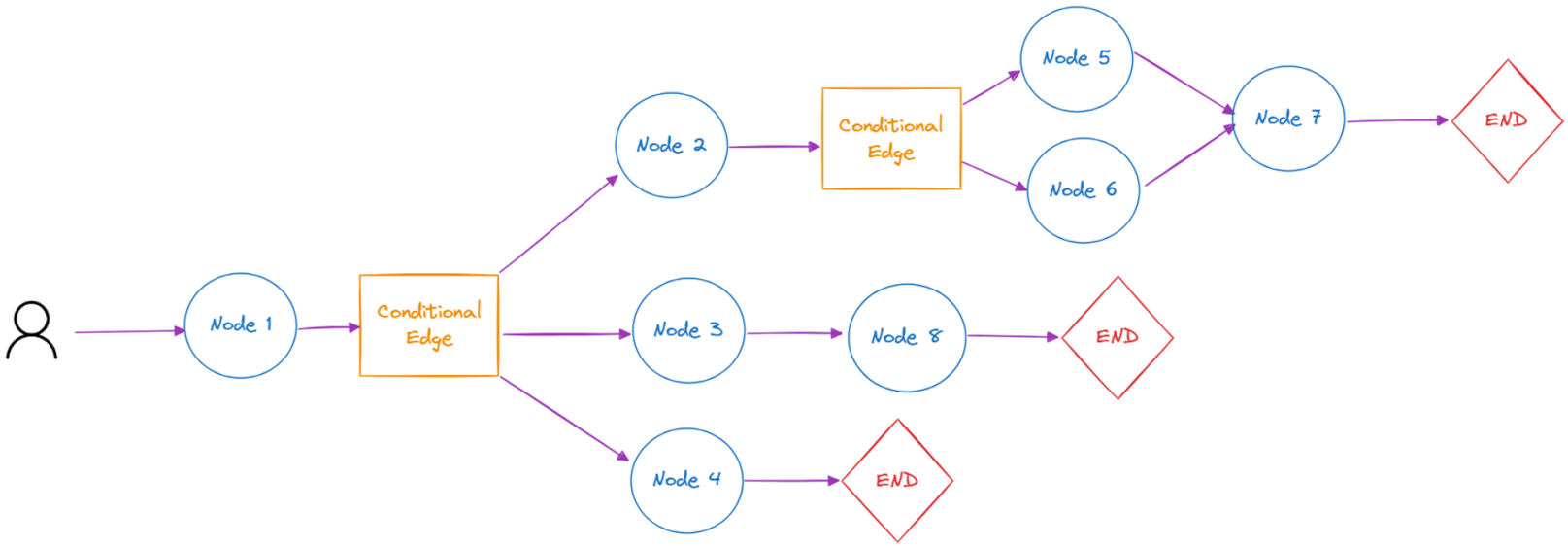
What Can Nodes Do? - Part 2

- **4. Tool-Calling Agents**:
  - Node can itself be an agent with tools
  - Example: Research node that uses search, calculator, Wikipedia tools
  - Inner agent makes tool decisions, outer graph manages workflow
- **5. Multi-Agent Nodes**:
  - Node can coordinate multiple sub-agents
  - Example: "Code Review" node with architect, tester, security agents
- **6. Any Computation**:
  - Data processing, calculations, file operations
  - Validation, formatting, business logic
- **Key Point**: Nodes are building blocks - combine them creatively

YHK

Core Components: Edges

- Connections between nodes in the graph
- Represent the flow of execution from one node to another
- Define possible paths through the workflow
- Can be simple directional connections
- Ensure proper sequence of operations



(Ref: A Comprehensive Guide About Langgraph: Code Included - Shivam Danawale)

YHK

Core Components: Conditional Edges

- Decision points in the workflow
- Enable branching based on specific conditions
- Example: After generation, route to criticism OR end based on quality
- Represented by dotted lines in diagrams
- Allow for dynamic execution paths
- Essential for implementing loops and conditional branching

YHK

Core Components: State

- A central object updated over time by the nodes in the graph
- Maintains context throughout workflow execution
- Preserves information between node executions
- State contains: messages, intermediate results, iteration count, tool outputs
- Enables nodes to access and modify shared information
- Critical for maintaining workflow coherence
- Supports complex, stateful operations

YHK

LangGraph Key Features

- **Looping and Branching**: Conditional statements and loop structures
- **State Persistence**: Automatic save/restore, pause and resume
- **Human-in-the-Loop**: Insert human review, state editing capabilities
- **Streaming Processing**: Real-time feedback on execution status
- **LangChain Integration**: Reuses existing components, LCEL support
- Provides controlled flexibility unlike pure React agents
- Production-ready with debugging and observability

YHK

Tutorial: Workflow Automation

Tutorial: Email Classification Workflow

- **Goal**: Automate email triage with classification and routing
- **Workflow Steps**:
  1. Read incoming email
  2. Classify as spam/legitimate using LLM
  3. Route to spam handler or ham for response drafter
  4. Draft response for legitimate/ham emails
- Demonstrates conditional routing and structured workflow
- Shows how graphs handle if/else logic naturally

YHK

Installation

LangGraph requires Python 3.8 or later.

```
pip install -U langgraph

python -c "import langgraph; print(langgraph.__version__)"
```

Step 1: Define State

- The state holds all information passed between nodes.
- We use `TypedDict` for the state schema.
- `Annotated` with `add_messages` tells LangGraph to append to the `messages` list instead of overwriting it.

```python
from typing import List, Dict, Annotated
from typing_extensions import TypedDict
from langgraph.graph.message import add_messages

class EmailState(TypedDict):
    email_content: str
    is_spam: bool
    draft_response: str
    # Each time a node returns a "messages" key, it will be appended to this list
    messages: Annotated[List[Dict[str, str]], add_messages]
```

YHK

State Management with Reducer Functions

- In LangGraph, nodes should return a dictionary containing only the fields they've changed. LangGraph then updates the central state.
- By default, a new value for a key **overwrites** the old one.
- To accumulate or modify values instead of overwriting, we use **reducer functions**.
- We can define how state keys should be updated using Python's `Annotated` type.

```python
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
import operator

# Use add_messages to append to the list of messages
# Use operator.add to sum numbers
class AppState(TypedDict):
    messages: Annotated[list, add_messages]
    num_steps: Annotated[int, operator.add]

# A node can now return just the new message and a number to add
def my_node(state: AppState):
    #... logic ...
    return {"messages": [("ai", "Hello!")], "num_steps": 1}
```

YHK

Step 2: Create Node Functions

```python
import os
from langchain_groq import ChatGroq

llm = ChatGroq(model="llama-3.1-8b-instant")

# This node doesn't modify the state, so it can just return an empty dictionary
def read_email(state: EmailState) -> dict:
    return {}

def classify_email(state: EmailState) -> dict:
    prompt = f"Classify this email as spam or legitimate:\n{state['email_content']}\nRespond with only 'spam' or 'legitimate'."
    response = llm.invoke(prompt)
    is_spam = "spam" in response.content.lower()
    # Return ONLY the fields that have changed
    return {
        "is_spam": is_spam,
        "messages": [{"role": "classifier", "content": response.content}]
    }
```

Step 2: Create Node Functions

```python
def handle_spam(state: EmailState) -> dict:
    # Return ONLY the new message to be added to the state
    return {
        "messages": [{"role": "system", "content": "Email marked as spam and moved to junk."}]
    }

def draft_response(state: EmailState) -> dict:
    prompt = f"Draft a professional response to:\n{state['email_content']}"
    response = llm.invoke(prompt)
    # Return ONLY the changed fields
    return {
        "draft_response": response.content,
        "messages": [{"role": "drafter", "content": response.content}]
    }
```

Step 3: Define Routing Logic

- Routing function decides next node based on state
- Returns the name of the target node
- Enables dynamic workflow execution

```python
def route_email(state: EmailState) -> str:
    """Conditional routing based on classification"""
    if state.is_spam:
        return "spam_handler"
    else:
        return "response_drafter"
```

YHK

Step 4: Build the Graph with Conditional Routing

- Conditional edges use a function to determine the next node based on the current state.
- The third argument to `add_conditional_edges` is a **path map**: a dictionary that maps the function's string output to the corresponding node name.

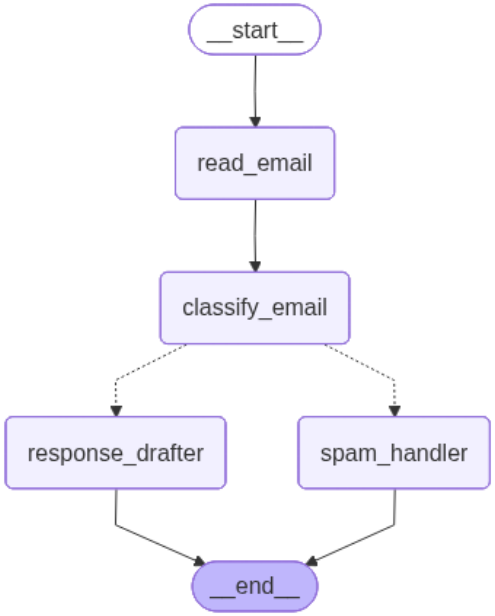
```python
from langgraph.graph import StateGraph, START, END

workflow = StateGraph(EmailState)

workflow.add_node("read_email", read_email)
workflow.add_node("classify_email", classify_email)
workflow.add_node("spam_handler", handle_spam)
workflow.add_node("response_drafter", draft_response)

workflow.set_entry_point("read_email")
workflow.add_edge("read_email", "classify_email")

def route_email(state: EmailState) -> str:
    if state["is_spam"]:
        return "spam"
    else:
        return "not_spam"

# The routing function's output ("spam" or "not_spam") is mapped to a node name
workflow.add_conditional_edges(
    "classify_email",
    route_email,
    {
        "spam": "spam_handler",
        "not_spam": "response_drafter"
```

YHK

Step 5: Visualize the Graph

- Visualize the workflow structure in ASCII format
- Shows nodes and their connections clearly

```
# Visualize and save the graph
png_graph = app.get_graph().draw_mermaid_png()
with open("langgraph_email_workflow_graph.png", "wb")
     as f:
    f.write(png_graph)
print(f"Graph saved as
     'langgraph_email_workflow_graph.png' in
     {os.getcwd()}")
```



Step 6: Run the Workflow

- Workflow executes automatically with proper routing
- State carries results through entire execution

```
# Test with legitimate email
state = EmailState(
    email_content="Hello, I'd like to schedule a meeting to discuss the
     project timeline."
)
result = app.invoke(state)
print(f"Is Spam: {result['is_spam']}")
print(f"Draft Response: {result['draft_response']}")

# Test with spam email
spam_state = EmailState(
    email_content="URGENT! You've won $1M! Click here now!!!"
)
spam_result = app.invoke(spam_state)
print(f"Is Spam: {spam_result['is_spam']}")

```

YHK

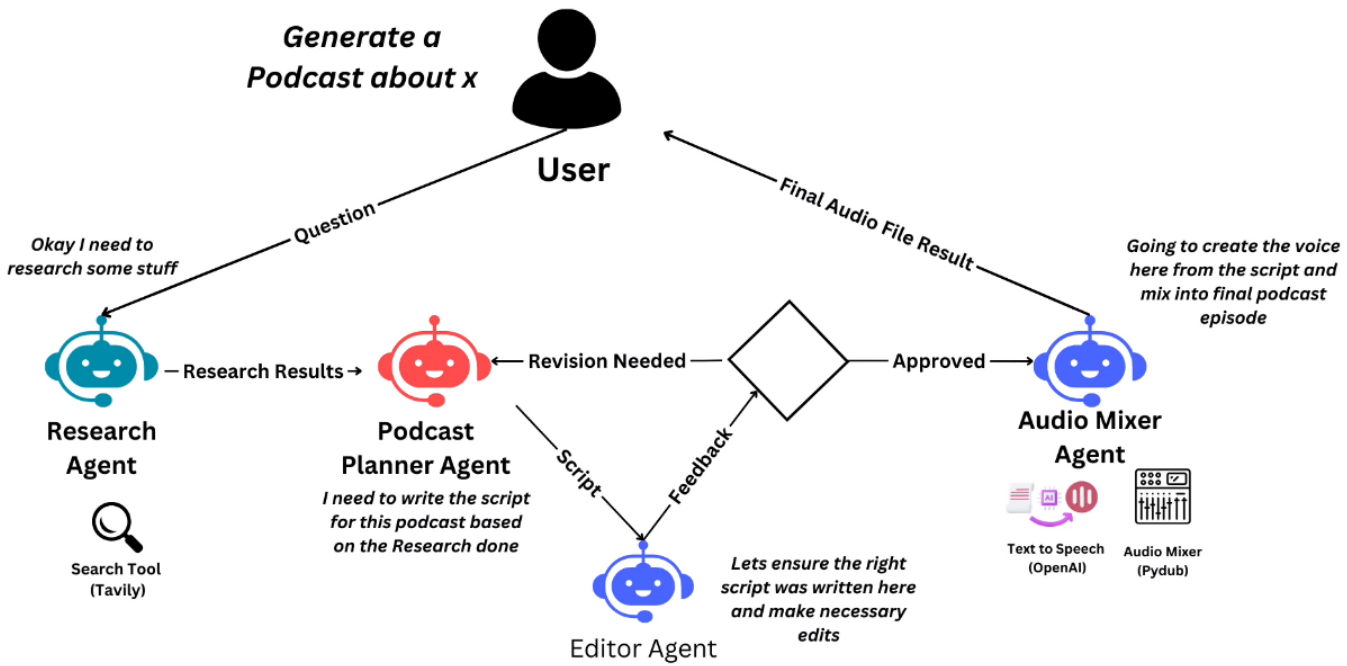
Tutorial Takeaways

- **Clear Structure**: Graph makes workflow explicit and understandable
- **Conditional Logic**: Natural if/else through routing functions
- **State Management**: Single state object tracks entire process
- **Extensibility**: Easy to add new nodes (e.g., "urgent" category)
- **Debugging**: Can inspect state at each node
- **Production-Ready**: Add checkpointing, error handling, retries
- This pattern scales to complex multi-step workflows

YHK

Real-World Applications

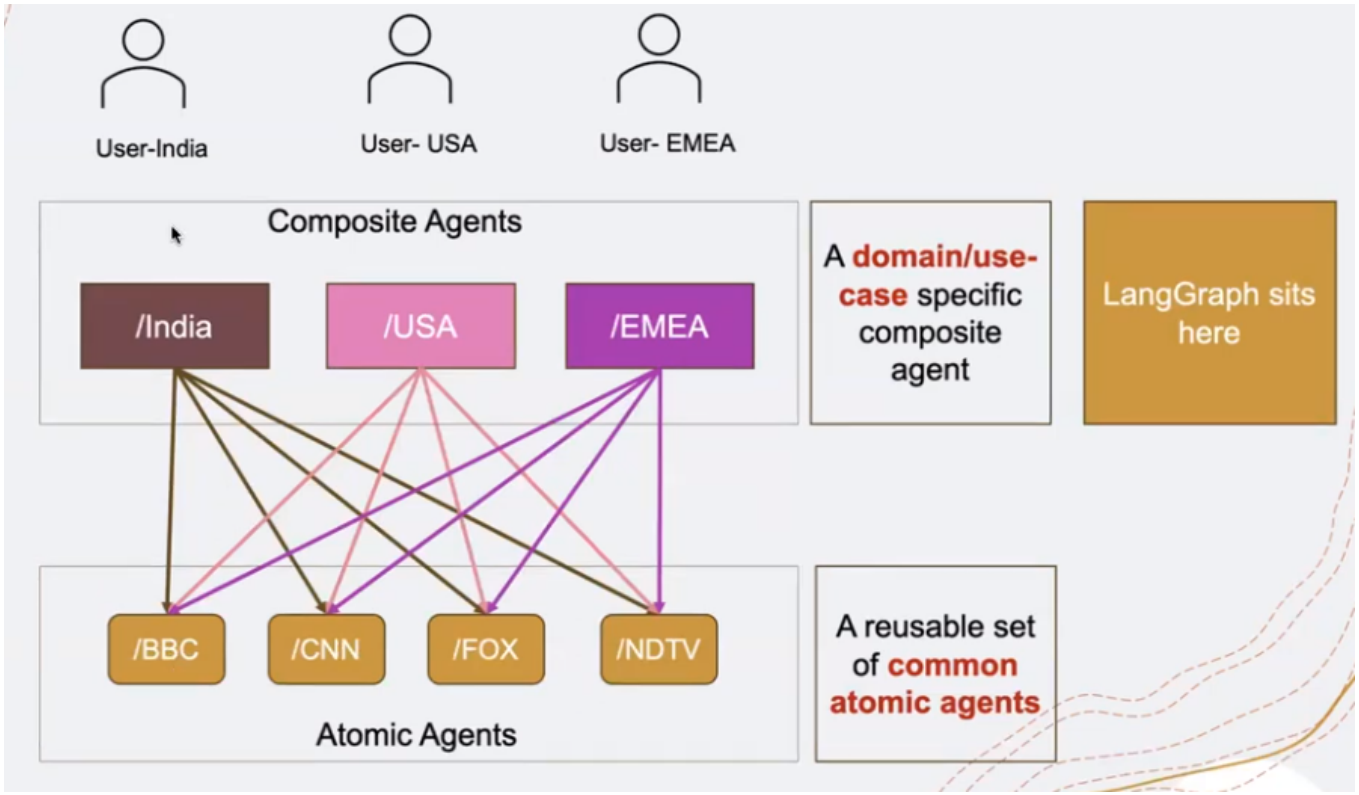
Example: Podcast Generator



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

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

Example: News Aggregator



Code:
https://github.com/rajib76/multi_agent/01_how_to_langgraph_example_01.py

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

YHK

Best Practices

- **State Design**: Keep simple and clear, use type hints, only necessary information
- **Node Functions**: Single responsibility, handle exceptions, return new state objects
- **Edge Design**: Clear conditional logic, avoid complex cycles, consider all paths
- **Error Handling**: Add at critical nodes, provide fallback mechanisms, log errors
- **Testing**: Test individual nodes, test routing logic, test complete workflows
- **Observability**: Use tracing tools (Langfuse, LangSmith) for production monitoring

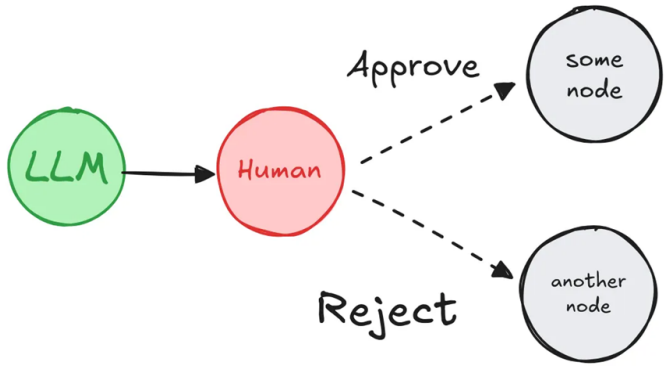
YHK

Advanced Concepts in LangGraph

YHK

Human-in-the-Loop: Overview

- **What**: Ability to pause execution and request human input/approval
- **Why Needed**:
  - Critical decisions require human judgment
  - Verify agent actions before execution
  - Edit/correct agent outputs
  - Ensure safety and compliance
- **Use Cases**:
  - Approve before sending emails or making purchases
  - Review generated code before deployment
  - Validate data modifications
  - Content moderation and quality control
- **LangGraph Implementation**: Interrupts and checkpointing



Human-in-the-Loop: Implementation Patterns

- **Pattern 1: Breakpoints**:
  - Pause execution at specific nodes
  - Example: Stop before "send_email" node
- **Pattern 2: Approval Gates**:
  - Node requests approval, waits for response
  - Conditional edge based on approval status
- **Pattern 3: State Editing**:
  - Pause, allow human to modify state
  - Resume with corrected state
- **Pattern 4: Continuous Monitoring**:
  - Human can interrupt at any time
  - Useful for long-running agents

YHK

Human-in-the-Loop: Code Example

```python
from langgraph.graph import StateGraph

# Create graph with checkpointing (required for interrupts)
memory = MemorySaver()
workflow = StateGraph(State)

workflow.add_node("draft_email", draft_email_node)
workflow.add_node("send_email", send_email_node)
workflow.add_edge("draft_email", "send_email")
# Compile with interrupt BEFORE send_email node
app = workflow.compile(
    checkpointer=memory,
    interrupt_before=["send_email"])  # Pause here for human approval
```

Human-in-the-Loop: Code Example

```
# Run workflow
config = {"configurable": {"thread_id": "1"}}
result = app.invoke(initial_state, config)

# At this point, execution is paused
# Human reviews the drafted email in result.draft
# To resume after approval:

app.invoke(None, config)  # Continues from where it stopped

# To modify state and resume:
updated_state = result.copy()
updated_state.draft = "Modified email content"
app.invoke(updated_state, config)

```

Advanced Concept: State Persistence and Checkpointing

- **What**: Automatically save state at each node execution
- **Benefits**:
  - Resume after failures or interruptions
  - Time-travel debugging (replay from any checkpoint)
  - Support long-running workflows across sessions
  - Enable human-in-the-loop patterns
- **Checkpoint Storage Options**:
  - In-memory (development/testing)
  - SQLite (local persistence)
  - PostgreSQL (production)
  - Redis (distributed systems)
- **Key Feature**: Each checkpoint is immutable and versioned
- Can fork from any checkpoint to explore alternative paths

YHK

Advanced Concept: Streaming and Real-Time Updates

- **What**: Stream intermediate results as graph executes
- **Why Important**:
  - Provide real-time feedback to users
  - Show progress in long-running workflows
  - Better user experience (vs waiting for completion)
- **Streaming Modes**:
  - **values**: Stream complete state after each node
  - **updates**: Stream only state changes
  - **messages**: Stream LLM token by token
- **Use Case**: Chatbot showing "thinking..." $\rightarrow$ "searching..." $\rightarrow$ "responding..."

YHK

Streaming Example

- Streaming provides transparency into agent execution
- Critical for production applications with users waiting

```python
# Instead of invoke(), use stream()
for chunk in app.stream(initial_state, config):
    # chunk contains state updates after each node
    print(f"Node: {chunk['node']}")
    print(f"State: {chunk['state']}")

# For token-by-token LLM streaming
async for event in app.astream_events(initial_state, config):
    if event["event"] == "on_chat_model_stream":
        # Stream each token as LLM generates
        print(event["data"]["chunk"], end="", flush=True)

```

YHK

Advanced Concept: Subgraphs and Modularity

- **What**: Embed complete graphs as nodes within parent graphs
- **Benefits**:
  - Modular, reusable workflow components
  - Hierarchical organization of complex systems
  - Encapsulation and separation of concerns
- **Example Hierarchy**:
  - Parent: Customer service orchestrator
  - Subgraph 1: Email classification workflow
  - Subgraph 2: Ticket routing workflow
  - Subgraph 3: Response generation workflow
- Each subgraph is self-contained with own state and logic
- Parent graph coordinates between subgraphs
- Enables building complex systems from simple components

YHK

Advanced Concept: Parallel Execution

- **What**: Execute multiple nodes simultaneously
- **Use Cases**:
  - Call multiple APIs concurrently
  - Search multiple data sources in parallel
  - Generate multiple variations simultaneously
- **Implementation**: Send edges from one node to multiple nodes

```
# Parallel node execution
workflow.add_node("search_web", search_node)
workflow.add_node("search_db", database_node)
workflow.add_node("search_docs", documents_node)
workflow.add_node("aggregate", aggregate_results)
# Fan-out: one node to many
workflow.add_edge("start", "search_web")
workflow.add_edge("start", "search_db")
workflow.add_edge("start", "search_docs")
# Fan-in: many nodes to one
workflow.add_edge("search_web", "aggregate")
workflow.add_edge("search_db", "aggregate")
workflow.add_edge("search_docs", "aggregate")
```

YHK

Advanced Concept: Dynamic Graph Modification

- **What**: Modify graph structure during execution
- **Use Cases**:
  - Add nodes based on runtime conditions
  - Dynamically adjust workflow based on results
  - Create adaptive systems that evolve
- **Example Scenario**:
  - Research agent discovers new sub-topics
  - Dynamically creates specialized research nodes
  - Each node investigates a different aspect
- **Limitation**: Advanced feature, use with caution
- Most use cases handled by conditional edges
- True dynamic modification for special scenarios only

YHK

Advanced Patterns: Map-Reduce

- **Pattern**: Split work, process in parallel, combine results
- **Example - Document Summarization**:
  1. Split large document into chunks (map)
  2. Summarize each chunk in parallel (parallel processing)
  3. Combine summaries into final summary (reduce)
- **Implementation**:
  - Split node: Divides input into subtasks
  - Multiple processing nodes: Execute in parallel
  - Aggregation node: Combines results
- **Benefits**: Scalability, speed, handles large inputs
- Common in RAG systems for processing multiple documents

YHK

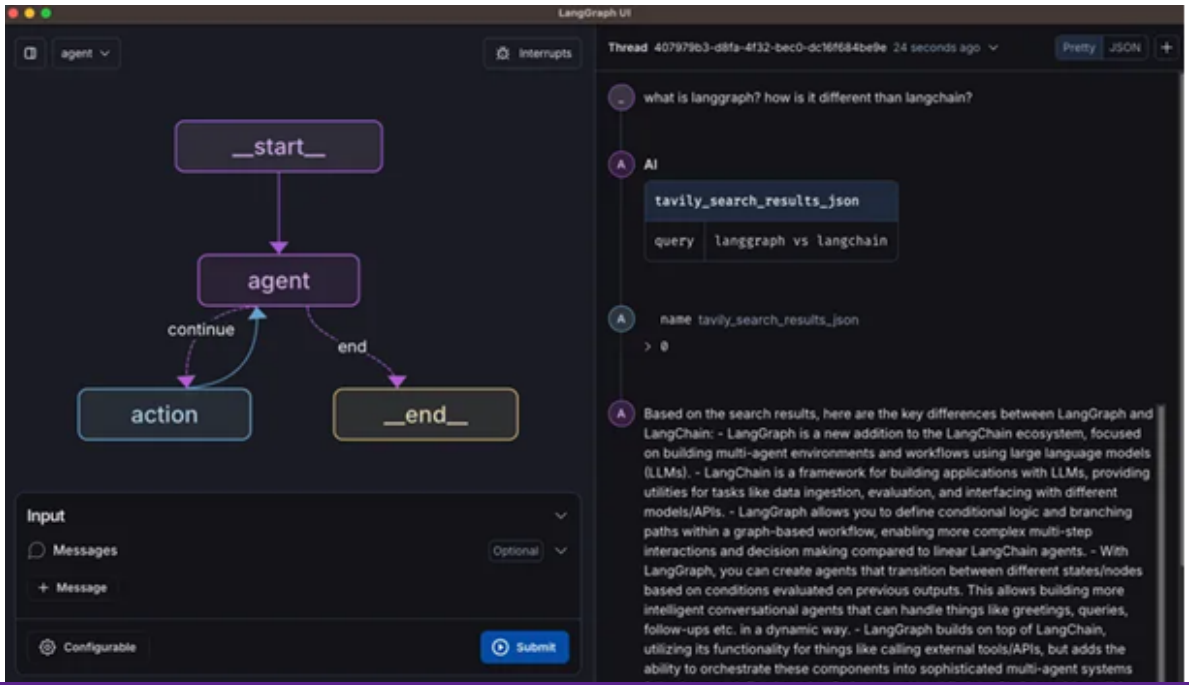
Best Practices for Advanced Features

- **Start Simple**: Add complexity only when needed
- **Checkpointing**: Always use for production systems
- **Error Handling**: Add try-except in nodes, fallback paths in graph
- **Observability**: Use LangSmith/Langfuse to trace execution
- **Testing**: Test nodes individually, then integration test graphs
- **State Size**: Keep state minimal - only essential data
- **Cycle Limits**: Set maximum iterations to prevent infinite loops
- **Human Approval**: For high-stakes actions (financial, emails, deletions)
- **Documentation**: Graph visualization helps team understanding

YHK

The LangGraph Ecosystem: LangGraph Studio

- LangGraph Studio is a visual development environment for building, debugging, and deploying LangGraph agents.
- It provides a **drag-and-drop interface** to construct complex graphs, making agent development more intuitive and accessible.
- Key Goals:
  - Lower the barrier to entry for building multi-agent systems.
  - Provide powerful debugging and observability tools.
  - Facilitate collaboration among developers.
  - Enable one-click deployment of agents.



Key Features of LangGraph Studio

- **Visual Editor**: Build and modify your agent's graph structure by dragging nodes and edges.
- **Real-time Debugging**: Step through your graph's execution, inspect the state at each node, and identify issues quickly.
- **Integrated LangSmith Tracing**: Seamlessly view detailed traces of your agent's runs, including LLM calls and tool usage.
- **Human-in-the-Loop**: A built-in UI allows for human review and intervention at any point in the graph.
- **Collaboration**: Share, review, and collaborate on agent designs with your team.
- **Deployment**: Deploy your agent as a scalable API endpoint directly from the studio.

YHK

Comparison: LangGraph vs. Other Agent Frameworks

Feature	LangGraph	CrewAI	AutoGen
Core Idea	A library for building stateful, multi-actor applications with graphs.	A framework for orchestrating role-playing autonomous AI agents.	A framework for simplified n versation workflows.
Control	**High**. Explicit control flow via graph structure. You define every possible path.	**Medium**. High-level process definition (e.g., sequential, hierarchical). Less granular control.	**Medium-Low**. Control is pr agent prompts and conversat
Architecture	Cyclic Graphs. Nodes are functions, edges define flow. Very flexible.	Agent-Task-Crew. Agents are given specific roles and tools; a process orchestrates them.	Conversation-based. Agents in a group chat, managed b manager.
Flexibility	**Very High**. Can build any custom flow, including human-in-the-loop and dynamic modification.	**Medium**. Optimized for common collaboration patterns but less flexible for arbitrary cycles.	**Medium**. Good for conversati complex, non-linear workflow model.
Best For	Production systems requiring reliability, control, and observability. Complex, long-running workflows.	Rapidly prototyping hierarchical or sequential agent teams for tasks like research or content creation.	Research and simulation of ag and conversational systems.

YHK

Production Deployment Considerations

- **Checkpointer Choice**:
  - `MemorySaver`: For development only.
  - `SqliteSaver`: Good for single-server, low-concurrency deployments.
  - `PostgresSaver` / `RedisSaver`: Essential for production. They support high concurrency and data resilience, allowing you to scale your application across multiple server instances.
- **Scalability**:
  - Use asynchronous node functions (`async def`) and invoke the graph with `ainvoke()` or `astream()` for better performance under load.
  - LangGraph's stateful nature allows you to run long-running tasks on background workers.
- **Monitoring & Observability**:
  - **LangSmith is critical**. It provides detailed, visual traces of every graph execution, making it possible to debug complex agent behavior, track costs, and monitor performance in production.
  - Implement logging within your nodes to capture key business events.

YHK

Summary

- LangGraph enables controlled, flexible AI workflows through graph structures
- Solves limitations of rigid chains and unpredictable pure agents
- Key components: Nodes (logic), Edges (flow), State (context)
- Ideal for workflow automation with intelligent decision-making
- Production-ready with state persistence, human-in-loop, streaming
- Start with simple workflows, expand to complex multi-agent systems
- The future of reliable, autonomous AI applications

YHK

Reflection Agents in LangGraph

(Ref: LangGraph Crash Course - Harish Neel)

YHK

Understanding Reflection - The Human Perspective

- Reflection means looking at yourself or your actions, like viewing yourself in a mirror
- Self-reflection after giving a presentation - thinking about how it went
- Re-reading an email after writing to ensure clarity and correctness
- Considering if a decision was the right choice after making it
- Having a mirror in front of you to think about improvements
- Thinking about what you've done in the past and how to do better
- Foundation concept for understanding AI reflection patterns

YHK

What is a Reflection Agent Pattern?

- AI system pattern that can look at its own outputs and improve them
- Similar to humans looking at themselves in a mirror and self-reflecting
- Makes AI systems better through iterative self-improvement
- Basic reflection agent system consists of two core components
- Generator agent - creates initial output or response
- Reflector agent - analyzes and critiques the generated output
- Both agents work together in a collaborative feedback loop
- Enables continuous improvement through multiple iterations

YHK

Practical Example: Viral Tweet Generator

- Simple application demonstrating basic reflection agent pattern
- Start node initiates the process with user input topic
- Tweet Generation Agent creates initial tweet based on provided topic
- Conditional edge routes output to Tweet Critiquing Agent
- Tweet Critiquing Agent acts as critical reviewer of generated content
- Critique focuses on viral potential: length, CTAs, hooks, hashtags
- Feedback loop continues for 4-6 iterations for improvement
- Each iteration produces progressively better, more viral-worthy tweets
- Real-world application potential as product feature for customers

YHK

System 1 vs System 2: Traditional vs Reflection Patterns

- **System 1 (Traditional):** One-shot prompt approach like ChatGPT/Claude
- Fast execution with immediate results
- Subconscious processing without active deep thinking
- Automatic response generation for everyday decisions
- More error-prone due to lack of reflection
- **System 2 (Reflection Pattern):** Two-agent collaborative approach
- Slower execution due to iterative feedback loops
- More conscious and effortful processing
- Better for complex decisions requiring precision
- Higher reliability through active thought-out responses

YHK

Reflection Loop Process Flow

- Initial generation phase creates first response/output
- Reflection phase analyzes generated content critically
- Reflector identifies merits: what went right in the output
- Reflector identifies issues: what went wrong and needs improvement
- Critique feedback sent back to generator brain/agent
- Generator incorporates feedback to revise and improve output
- Loop repeats for predetermined number of iterations (n times)
- Final improved output delivered to user after iteration completion
- Process applicable beyond tweets to any content generation task

YHK

Three Types of Reflection Agents in LangGraph

- **Basic Reflection Agents:** Foundation pattern with generator and reflector
- Reflector chain where reflector agent and generator agent work together
- Simple feedback loop for iterative improvement
- **Reflexion Agents:** Advanced pattern building on basic reflection
- Enhanced capabilities beyond basic reflection mechanisms
- **Language Agent Tree Search (LATS):** Most sophisticated approach
- Tree-based search methodology for complex decision making
- Recommended to master basic reflection first before advancing
- Progressive learning path from basic to advanced patterns

YHK

Basic Reflection Agents

Adding to graph

(Ref: LangGraph Crash Course - Harish Neel)

```python
from typing import List, Sequence
from langchain_core.messages import BaseMessage, HumanMessage
from langgraph.graph import END, StateGraph
from typing_extensions import TypedDict, Annotated
from langgraph.graph.message import add_messages
from chains import generation_chain, reflection_chain

# Define the state for our graph
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]

def generate_node(state: AgentState):
    return {"messages": generation_chain.invoke({"messages": state["messages"]})}

def reflect_node(state: AgentState):
    response = reflection_chain.invoke({"messages": state["messages"]})
    return {"messages": [HumanMessage(content=response.content)]}

graph_builder = StateGraph(AgentState)
graph_builder.add_node("generate", generate_node)
graph_builder.add_node("reflect", reflect_node)
graph_builder.set_entry_point("generate")
```

YHK

Basic Reflection Agents

Adding to graph

(Ref: LangGraph Crash Course - Harish Neel)

```
def should_continue(state: AgentState):
    if len(state['messages']) > 6:
        # End the loop after 3 iterations (1 initial + 2 reflections)
        return END
    return "reflect"

graph_builder.add_conditional_edges("generate", should_continue)
graph_builder.add_edge("reflect", "generate")

app = graph_builder.compile()

# Visualize the graph
print(app.get_graph().draw_ascii())

# Invoke the graph
response = app.invoke({"messages": [HumanMessage(content="AI Agents taking over content
      creation")]})

print(response['messages'][-1].content)
```

YHK

Structured LLM Outputs

Introduction to Structured Outputs

- LLMs traditionally return unstructured string responses that are difficult to process programmatically
- Structured outputs allow models to return data matching specific schemas we define
- Essential for software engineering where we work with objects, JSON, and database-ready formats
- Instead of random strings, get organized data with exact properties for manipulation
- Multiple output formats supported: JSON, dictionary, string, YAML, HTML
- Example: Request a joke and receive setup, punchline, and rating in structured JSON format
- Enables seamless integration with databases and application workflows
- Critical foundation for building advanced agentic AI systems like reflexion agents
- Transforms LLM responses from text blobs into actionable data structures
- Makes LLM outputs suitable for automated processing and storage systems

YHK

Pydantic Models for Structured Output

- Pydantic is a Python library that defines data structures as blueprints for data validation
- Uses Python type hints (string, integer) to enforce correct data types automatically
- Define classes with required fields and descriptions for each property
- Integrated with LangChain using `with_structured_output()` method for LLM binding
- Works with any chat model (OpenAI, Groq, Llama) by simply swapping the model
- Internally converts Pydantic model into a tool that LLM must use exclusively
- Automatic validation ensures all properties are present with correct data types
- Throws errors automatically if validation fails, enabling fallback strategies
- LLM receives tool schema with property descriptions and required field specifications
- Response parsing and conversion handled automatically by LangChain framework

YHK

Pydantic Model Implementation Example

- Use `with_structured_output()` to bind model to LLM instance
- Tool choice parameter forces LLM to use only the specified schema
- Request payload includes tools array with function type and parameter specifications
- Response contains structured JSON matching exact schema requirements
- LangChain handles parsing, validation, and Pydantic model instantiation automatically
- Validation failures trigger automatic error handling for robust applications

```python
from pydantic import BaseModel, Field

class Country(BaseModel):
    """Information about a country"""
    name: str = Field(description="Name of the country")
    language: str = Field(description="Language of the country")
    capital: str = Field(description="Capital of the country")

structured_llm = model.with_structured_output(Country)
response = structured_llm.invoke("Tell me about France")
```

Modern Tool Binding with `.bind_tools()`

- While `with_structured_output` is good for forcing a single schema, the modern approach for general tool use is `.bind_tools()`.
- This method attaches one or more tools (Pydantic models or functions) to an LLM.
- The model can then decide which tool to call, if any, based on the input prompt. This is the foundation of most agentic behavior.

```python
from langchain_openai import ChatOpenAI
from langchain_core.pydantic_v1 import BaseModel, Field

# Define a tool schema
class GetWeather(BaseModel):
    """Get the current weather in a given location"""
    location: str = Field(description="The city and state, e.g. San Francisco, CA")

# Initialize the model
llm = ChatOpenAI(model="gpt-4o")

# Bind the tool to the model
llm_with_tools = llm.bind_tools([GetWeather])

# Invoke the model
ai_msg = llm_with_tools.invoke("what is the weather in Boston?")

# The output contains a 'tool_calls' attribute
print(ai_msg.tool_calls)
# > [{'name': 'GetWeather', 'args': {'location': 'Boston'}, 'id': '...'}]
```

YHK

Alternative Methods: TypedDict and JSON Schema

- TypedDict approach uses Python typing without Pydantic validation overhead
- Annotated class provides type hints, default values, and property descriptions
- JSON schema method offers direct schema definition without Python classes
- Schema requires title, description, properties, and required field specifications

```python
# TypedDict approach
from typing import TypedDict, Annotated, Optional

class JokeDict(TypedDict):
    setup: Annotated[str, "Setup of the joke"]
    punchline: Annotated[str, "Punchline to the joke"]
    rating: Annotated[Optional[int], None, "Joke rating 1-10"]

# JSON Schema approach
joke_schema = {"title": "Joke","type": "object",
    "properties": {
        "setup": {"type": "string", "description": "Setup of joke"},
        "punchline": {"type": "string", "description": "Punchline"}
    },"required": ["setup", "punchline"]
}
```

YHK

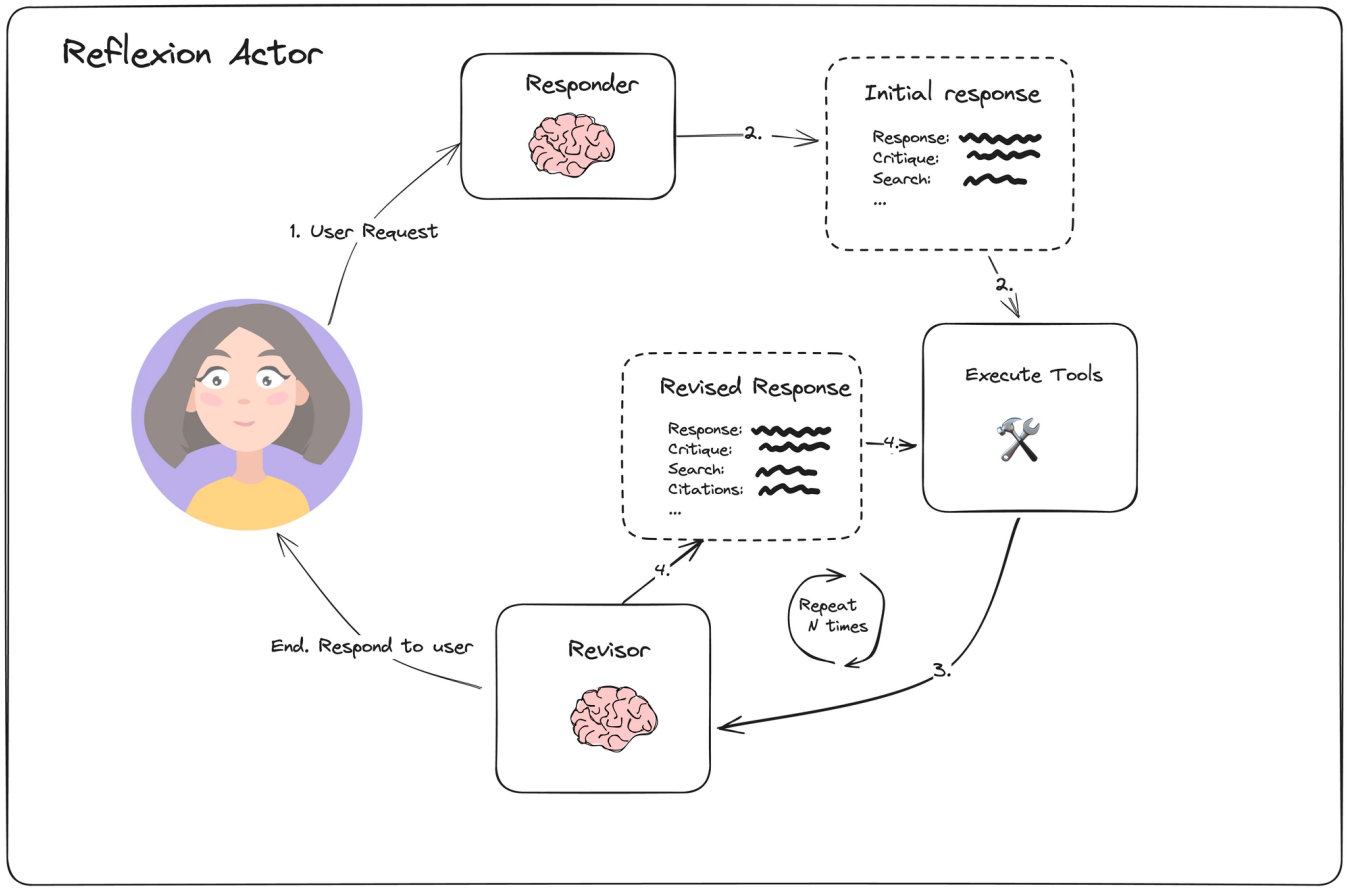
Reflexion Agent in LangGraph

Reflexion Agent System: Overview

- Addresses drawbacks of basic reflection systems that lack grounding in live data
- Unlike reflection agents, reflexion agents fact-check with external data via API calls
- Combines self-critique with internet search capabilities for enriched content
- Generates citations and references from external sources during content creation
- Particularly useful for marketers writing blog posts requiring recent information
- Not limited to LLM training data cutoff dates like traditional reflection systems
- Iteratively improves content quality through multiple search-revise cycles
- Enables creation of well-researched, factually grounded content with live data
- Prevents hallucinations by validating information against current sources
- Supports episodic memory for context-aware, personalized interactions over time

YHK

Reflexion Agent



(Ref: LangGraph Crash Course - Harish Neel)

Core Components Architecture

- **Actor Agent:** Main driver that orchestrates all subcomponents and processes
- **Responder Agent:** Generates initial response with self-reflection capabilities
- **Revisor Agent:** Creates revised responses based on new external data
- **Tools/Tool Execution:** Handles API calls, primarily internet search functionality
- **Episodic Memory:** Recalls specific past interactions and experiences for context
- Actor contains all subcomponents: responder, revisor, and execute tools modules
- System operates through iterative loops of generation, search, and revision
- Each component has specific responsibilities within the overall workflow
- Tool execution primarily uses Tavily Search for gathering live internet data
- Architecture designed for modularity and extensibility with additional tools

YHK

Workflow and Data Flow

- User request goes to responder agent for initial content generation
- Responder outputs JSON with response, critique, and suggested search keywords
- Execute tools component searches internet using suggested keywords via Tavily
- Search results combined with initial response fed to revisor agent
- Revisor generates revised response with updated critique and citations
- Citations added to reference external sources used in content creation
- Process iterates: revised search terms trigger new searches and revisions
- Each iteration enriches content with more recent, relevant information
- Content length maintained (e.g., 250 words) while improving quality and accuracy
- Final output returned to user after multiple refinement cycles complete

YHK

JSON Response Structure Example

- Responder agent outputs structured JSON with three main properties
- **Response:** Contains the initial generated content (e.g., 250-word blog post)
- **Critique:** Self-assessment identifying areas for improvement in content
- **Search Terms:** Keywords suggested for gathering additional relevant information
- Revisor agent extends structure with additional citation property
- **Citations:** References to external sources with URLs and factual backing
- Revised versions update all properties: response, critique, search terms, citations
- Structure enables systematic improvement through multiple iterations
- JSON format facilitates programmatic processing and integration with LangGraph
- Supports grounding of content claims with verifiable external sources

YHK

Advanced Concepts in LangGraph

YHK

Custom Node Types

Create custom node types to implement complex agent logic. This provides flexibility and control over your application's behavior.

```python
from typing import Annotated
from langchain_anthropic import ChatAnthropic
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
class MyCustomNode:
    def __init__(self, llm):
        self.llm = llm
    def __call__(self, state):
        # Implement your custom logic here
        # Access the state and perform actions
        messages = state["messages"]
        response = self.llm.invoke(messages)
        return {"messages": [response]}
graph_builder = StateGraph(State)
llm = ChatAnthropic(model="claude-3-haiku-20240307")
custom_node = MyCustomNode(llm)
graph_builder.add_node("custom_node", custom_node)
```

YHK

Advanced Conditional Edges

Conditional edges allow to dynamically decide the execution flow based on the state.

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

class AgentState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    tools_output: Dict[str, str] = {}
    status: str = "RUNNING"
    error_count: int = 0

def route_by_status(state: AgentState) -> Literal["process", "retry", "error", "end"]:
    if state.status == "SUCCESS":
        return "end"
    elif state.status == "ERROR":
        if state.error_count >= 3:
            return "error"
        return "retry"
    elif state.status == "NEED_TOOL":
        return "process"
    return "process"

workflow.add_conditional_edges(
    "check_status",
    route_by_status,
    {"process": "execute_tool", "retry": "retry_handler",
     "error": "error_handler", "end": END}
)
```

YHK

State management

LangGraph offers powerful state management techniques, which include using external databases like SQLite, PostgreSQL, and MongoDB, or cloud storage solutions like Amazon S3, Google Cloud Storage, and Azure Blob Storage to store and retrieve your agent's state, enabling reliability and scalability.
Here's an example of using a SQLite database for state management:

```
from langgraph.checkpoint.sqlite import SqliteSaver
# Connect to the SQLite database
memory = SqliteSaver.from_conn_string(":memory:")
# Compile the graph with the checkpointer
graph = graph_builder.compile(checkpointer=memory)
```

YHK

Advanced State Management: Checkpointing

- **Checkpointing** is the mechanism for saving the state of a graph at each step. It is the foundation for persistence, resilience, and human-in-the-loop interactions.
- **Benefits**:
  - **Persistence**: Stop and resume long-running workflows across sessions.
  - **Resilience**: Recover from failures without losing progress.
  - **Debugging**: "Time-travel" by rolling back to a previous state to inspect it.
  - **Human-in-the-Loop**: Pause the graph, await human input, and then resume.
- **Backends**: LangGraph supports various storage backends for checkpoints.

```python
from langgraph.checkpoint.sqlite import SqliteSaver

# In-memory saver for development (not persistent)
# memory = MemorySaver()

# SQLite saver for local persistence
memory = SqliteSaver.from_conn_string("my_agent_state.db")

# Compile the graph with the checkpointer
# A unique thread_id is used to track conversations
config = {"configurable": {"thread_id": "user-123"}}
app = graph.compile(checkpointer=memory)

# Run the graph - state is saved automatically
app.invoke({"messages": [("human", "Hi!")]}, config)
```

YHK

Interacting with Checkpoints

- Once a checkpointer is configured, you can interact with the state of any conversation (thread).
- This allows you to inspect, modify, and travel through the history of a workflow.

```python
# The config identifies which conversation thread to interact with
config = {"configurable": {"thread_id": "user-123"}}

# Get the latest state of the conversation
current_state = app.get_state(config)
print(current_state)
# -> {'messages': [HumanMessage(...), AIMessage(...)]}

# Get the entire history of states (snapshots after each step)
history = app.get_state_history(config)
print(f"History has {len(history)} steps.")

# Update the state directly (e.g., to inject a correction)
app.update_state(
    config,
    {"messages": [("user", "Actually, I have a different question.")]}
)

# You can also roll back to a previous state by passing a specific checkpoint
# from the `get_state_history` call to `update_state`.
```

Error handling

LangGraph also provides mechanisms for error handling:

- Exceptions: Node functions can raise exceptions to signal errors during execution. You can catch and handle these exceptions to prevent your graph from crashing.
- Retry mechanisms: You can implement retry logic within your nodes to handle transient errors, such as network issues or API timeouts.
- Logging: Use logging to record errors and track the execution of your graph.

YHK

Error Handling Patterns

- **Robust agents must handle unexpected failures**, like API errors or invalid LLM outputs.
- **Pattern 1: Try/Except in Nodes**: The simplest method is to wrap node logic in a `try...except` block. The node can then return an error message in the state.
- **Pattern 2: Fallback Edges**: Use conditional edges to create dedicated error-handling paths. If a node fails, it can update the state with an 'ERROR' status, which a router function can use to divert the flow to a recovery or reporting node.
- **State Tracking**: Use the state to track error counts. After a certain number of retries, the graph can route to a final failure node.

```
def tool_node(state):
    try:
        # ... call an external API ...
        result = ...
        return {"result": result, "status": "SUCCESS"}
    except Exception as e:
        # On failure, update state with error info
        return {"error_message": str(e), "status": "ERROR"}

def router(state):
    if state.get("status") == "ERROR":
        return "error_handler_node"
    return "next_step_node"

```

Testing LangGraph Applications

- **Unit Testing Nodes**:
  - Nodes are just Python functions. You can test them in isolation by creating a sample input state and asserting the output dictionary is correct.
  - This allows you to verify the logic of each step independently.
- **Integration Testing the Graph**:
  - Test the compiled graph (`app`) by invoking it with a representative input.
  - Assert that the final state is what you expect. This tests the interaction between nodes and the correctness of the routing logic.
  - Use `app.stream()` to check the state at intermediate steps.
- **Mocking**: Use mocking libraries (like `unittest.mock`) to replace external dependencies like LLMs or APIs during tests. This makes tests faster, cheaper, and more predictable.

YHK

Parallel Execution in LangGraph

Supports parallel execution of multiple nodes, which is particularly useful for handling complex tasks:

```python
async def parallel_tools_execution(state: AgentState) -> AgentState:
    """Parallel execution of multiple tools"""
    tools = identify_required_tools(state.current_input)

    async def execute_tool(tool):
        result = await tool.ainvoke(state.current_input)
        return {tool.name: result}

    # Execute all tools in parallel
    results = await asyncio.gather(*[execute_tool(tool) for tool in tools])

    # Merge results
    tools_output = {}
    for result in results:
        tools_output.update(result)

    return AgentState(
        messages=state.messages,
        current_input=state.current_input,
        tools_output=tools_output,
        status="SUCCESS"
    )

```

ReACT Architecture Introduction

- ReACT (Reasoning and Acting) combines reasoning and acting capabilities
- Agent solves problems through continuous cycle: Reason $\rightarrow$ Act $\rightarrow$ Observe
- Flexible response to complex tasks using external tools
- Enhanced capabilities through tool integration
- LangGraph provides pre-built ReACT agents
- Supports Google search, DALL-E image generation, and more
- Easy implementation with create_react_agent function
- Suitable for dynamic problem-solving scenarios

YHK

ReACT Agent Implementation

```python
import dotenv
from langchain_community.tools import GoogleSerperRun
from langchain_community.tools.openai_dalle_image_generation import OpenAIDALLEImageGenerationTool
from langchain_openai import ChatOpenAI
from langgraph.prebuilt.chat_agent_executor import create_react_agent

dotenv.load_dotenv()

# Define Tools and Parameter Schemas
google_serper = GoogleSerperRun(
    name="google_serper",
    description="A low-cost Google search API for current events",
    args_schema=GoogleSerperArgsSchema,
    api_wrapper=GoogleSerperAPIWrapper(),
)

dalle = OpenAIDALLEImageGenerationTool(
    name="openai_dalle",
    api_wrapper=DallEAPIWrapper(model="dall-e-3"),
    args_schema=DallEArgsSchema,
)

tools = [google_serper, dalle]
model = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# Create ReACT agent
agent = create_react_agent(model=model, tools=tools)

# Usage
result = agent.invoke({"messages": [("human", "Help me draw a shark flying in the sky")]})
```

YHK

Analysis of Results

- the agent will first understand the task requirements and then decide to use the DALL-E tool to generate an image.
- It will generate a detailed image description and then call the DALL-E API to create the image.
- Finally, it will return the generated image URL along with a brief description.
- The output might look like this:

```
{
  "messages": [
    HumanMessage(content='Help me draw a picture of a shark flying in the sky'),
    AIMessage(content='', additional_kwargs={'tool_calls': [...]}),
    ToolMessage(content='https://dalleproduse.blob.core.windows.net/...'),
    AIMessage(content='Here is the image you requested: a picture of a shark flying in the sky. You can view the image by
      clicking the link below.\n\n![Shark flying in the sky](https://dalleproduse.blob.core.windows.net/...)')
  ]
}
```

YHK

Message Management in LangGraph

- Message accumulation can lead to performance issues
- delete_messages function removes processed messages
- Filtering techniques control message flow
- Conditional edges for message routing
- Time-based and quantity-based pruning strategies
- Essential for long-running applications
- Maintains conversation clarity and focus
- Improves overall system performance

YHK

Message Deletion and Filtering

Can delete messages based on specified conditions. This function can be used in the nodes of the graph, especially after processing certain messages.

```python
from langgraph.prebuilt import ToolMessage, delete_messages

def process_and_delete(state):
    # Processing logic
    # Delete processed messages
    state = delete_messages(state, lambda x: isinstance(x, ToolMessage))
    return state

def filter_messages(state):
    filtered_messages = [msg for msg in state['messages']
                         if not isinstance(msg, ToolMessage)]
    return {"messages": filtered_messages}

def keep_latest_messages(state, max_messages=50):
    return {"messages": state['messages'][-max_messages:]}

# Time-based pruning
from datetime import datetime, timedelta

def prune_old_messages(state):
    current_time = datetime.now()
    recent_messages = [msg for msg in state['messages']
                       if current_time - msg.timestamp < timedelta(hours=1)]
    return {"messages": recent_messages}
```

Checkpoint Mechanism

- Checkpoints are snapshots during graph execution
- Enable pause and resume functionality for long-running tasks
- Useful for processes requiring human intervention
- Support state rollback to previous points
- Allow data modification at checkpoints
- Use thread_id and thread_ts for unique identification
- Retrieve last state and execution history
- Essential for resumable AI applications

YHK

Checkpoint Implementation

```python
from langgraph.checkpoint import create_checkpoint, load_checkpoint

def process_with_checkpoint(state):
    # Processing logic
    # Create a checkpoint
    checkpoint = create_checkpoint(state)
    return {"checkpoint": checkpoint, "state": state}

def resume_from_checkpoint(checkpoint):
    state = load_checkpoint(checkpoint)
    # Continue processing
    return state

# Using checkpoints in practice
def summarize_and_prune(state):
    summary = summarize_conversation(state['messages'])
    new_messages = state['messages'][-5:]
    new_messages.append(ToolMessage(content=summary))
    state['messages'] = new_messages

    # Create checkpoint
    checkpoint = create_checkpoint(state)
    state['checkpoint'] = checkpoint
    return state

# Retrieve state and history
graph.get_state(config)  # Get last saved state
graph.get_state_history(config)  # Get all saved states
```

Human-in-the-Loop Interaction

- Allows human participation in AI decision-making process
- Callback functions obtain human input during execution
- Conditional branching determines when human intervention needed
- Confidence-based routing for automatic vs manual processing
- Enhanced system reliability through human oversight
- Flexible interaction control mechanisms
- Improves decision quality in critical scenarios
- Supports collaborative human-AI workflows

YHK

Human Interaction Implementation

```python
def human_input_node(state):
    # Display current state to user
    print("Current state:", state)
    # Get user input
    user_input = input("Please provide your input: ")
    # Update state
    state['user_input'] = user_input
    return state

def check_confidence(state):
    if state['confidence'] < 0.8:
        return "human_input"
    else:
        return "auto_process"

def human_intervention(state):
    print("Current conversation:", state['messages'])
    human_response = input("Please provide assistance: ")
    state['messages'].append(HumanMessage(content=human_response))
    return state

# Add conditional routing
graph.add_conditional_edges("process_query", {
    "human_intervention": lambda s: s['confidence'] < 0.8,
    "auto_process": lambda s: s['confidence'] >= 0.8
})
```

Subgraph Architecture

A subgraph is essentially a complete graph structure that can be used as a node in a larger graph structure.

- Break complex workflows into manageable components
- Modular design enhances code reusability
- Independent subgraphs improve maintainability
- Easy testing and debugging of individual components
- Scalable architecture for adding new features
- Encapsulate complex logic in reusable modules
- Support composition and interaction between subgraphs
- Enable hierarchical workflow organization

YHK

Subgraph Implementation

Creating a Basic Subgraph

```
from langgraph.graph import SubGraph, Graph

class ContentGenerationSubGraph(SubGraph):
    def build(self) -> Graph:
        graph = Graph()

        def generate_content(state):
            # Content generation logic
            return state

        def review_content(state):
            # Content review logic
            return state
```

Subgraph Implementation

State Management in Subgraph

```
class AnalyticsSubGraph(SubGraph):
    def build(self) -> Graph:
        graph = Graph()

        def process_analytics(state):
            # Ensure the state contains necessary keys
            if 'metrics' not in state:
                state['metrics'] = {}
            # Process analytics data
            state['metrics']['engagement'] = calculate_engagement(state)
            return state

        graph.add_node("analytics", process_analytics)
        return graph
```

YHK

Subgraph Implementation

Using Subgraphs in the Main Graph

```
def create_marketing_workflow():
    main_graph = Graph()
    # Instantiate subgraphs
    content_graph = ContentGenerationSubGraph()
    analytics_graph = AnalyticsSubGraph()
    # Add subgraphs to the main graph
    main_graph.add_node("content", content_graph)
    main_graph.add_node("analytics", analytics_graph)
    # Connect subgraphs
    main_graph.add_edge("content", "analytics")
    return main_graph
```

YHK

Subgraph Implementation

Data Passing Between Subgraphs

```
class DataProcessingSubGraph(SubGraph):
    def build(self) -> Graph:
        graph = Graph()

        def prepare_data(state):
            # Prepare data for use by other subgraphs
            state['processed_data'] = {
                'content_type': state['raw_data']['type'],
                'metrics': state['raw_data']['metrics'],
                'timestamp': datetime.now()
            }
            return state

        graph.add_node("prepare", prepare_data)
        return graph
```

Practical Case: Implementation of Marketing Agent

Content Generation Subgraph

```
class ContentCreationSubGraph(SubGraph):
    def build(self) -> Graph:
        graph = Graph()

        def generate_content(state):
            prompt = f"""
            Target Audience: {state['audience']}
            Platform: {state['platform']}
            Campaign Goal: {state['goal']}
            """
            # Use LLM to generate content
            content = generate_with_llm(prompt)
            state['generated_content'] = content
            return state

        def optimize_content(state):
            # Optimize content according to platform characteristics
            optimized = optimize_for_platform(state['generated_content'], state['platform'])
            state['final_content'] = optimized
            return state

        graph.add_node("generate", generate_content)
        graph.add_node("optimize", optimize_content)
        graph.add_edge("generate", "optimize")
        return graph
```

YHK

Practical Case: Implementation of Marketing Agent

Analytics Subgraph

```
class AnalyticsSubGraph(SubGraph):
    def build(self) -> Graph:
        graph = Graph()

        def analyze_performance(state):
            metrics = calculate_metrics(state['final_content'])
            state['analytics'] = {
                'engagement_score': metrics['engagement'],
                'reach_prediction': metrics['reach'],
                'conversion_estimate': metrics['conversion']
            }
            return state

        def generate_recommendations(state):
            recommendations = generate_improvements(state['analytics'], state['goal'])
            state['recommendations'] = recommendations
            return state

        graph.add_node("analyze", analyze_performance)
        graph.add_node("recommend", generate_recommendations)
        graph.add_edge("analyze", "recommend")
        return graph
```

YHK

Practical Case: Implementation of Marketing Agent

Main Workflow

```
def create_marketing_agent():
    main_graph = Graph()
    # Instantiate subgraphs
    content_graph = ContentCreationSubGraph()
    analytics_graph = AnalyticsSubGraph()

    # Add configuration node
    def setup_campaign(state):
        # Initialize marketing campaign configuration
        if 'config' not in state:
            state['config'] = {
                'audience': state.get('audience', 'general'),
                'platform': state.get('platform', 'twitter'),
                'goal': state.get('goal', 'engagement')
            }
        return state

    main_graph.add_node("setup", setup_campaign)
    main_graph.add_node("content", content_graph)
    main_graph.add_node("analytics", analytics_graph)
    # Build workflow
    main_graph.add_edge("setup", "content")
    main_graph.add_edge("content", "analytics")
    return main_graph
```

Subgraph Best Practices and Considerations

- Design Principles for Subgraphs
  - Keep subgraph functionality singular
  - Ensure clear input and output interfaces
  - Properly handle state passing
- Performance Considerations
  - Avoid frequent large data transfers between subgraphs
  - Design state storage structures reasonably
  - Consider asynchronous processing needs
- Error Handling
  - Implement error handling within subgraphs
  - Provide clear error messages
  - Ensure state consistency

YHK

Data State and Induction Functions

- Default behavior overwrites original data completely
- Manual state retrieval and update prevents data loss
- Induction functions provide automatic data accumulation
- Annotated wrapper simplifies state management
- Independent node execution without state conflicts
- Type hints improve code clarity and debugging
- Simplified node modification when updating structures
- Enhanced data consistency across workflow execution

YHK

Induction Functions Example

Understanding how data states are handled in LangGraph is crucial. By default, the dictionary data returned by nodes will overwrite the original data, which may lead to unexpected results. For example:

```python
from typing import TypedDict, Annotated

# Problem: Default behavior overwrites data
class MyState(TypedDict):
    messages: list

def fn1(state: MyState):
    return {"messages": [4]}

r = graph.invoke({"messages": [1, 2, 3]})
# Result: {"messages": [4]} instead of [1,2,3,4]
```

YHK

Induction Functions Example

To solve this problem, LangGraph provides two methods for accumulating data:

```
# Manually retrieve and update the original state:
 def fn1(state: MyState):
    old = state.get("messages", [])
    return {"messages": old + [4]}


# Use LangGraph's Annotated wrapper and induction functions:
def concat_lists(original: list, new: list) -> list:
    return original + new

class MyState(TypedDict):
    messages: Annotated[list, concat_lists]

def fn1(state: MyState):
    return {"messages": [4]}

r = graph.invoke({"messages": [1, 2, 3]})
# Result: {'messages': [1, 2, 3, 4]}

```

Parallel Node Execution

- END node signifies route termination, not graph termination
- Nodes at same level execute in parallel
- Execution order is uncertain in parallel execution
- Control flow by adjusting node connections
- Important for understanding graph execution model
- Optimize performance through parallel processing
- Design considerations for concurrent operations
- Manage dependencies between parallel nodes

```
1 graph.add_edge(["left1", "right3"], "merge")
```

YHK

CheckPoint Mechanism

Checkpoints can be seen as a storage medium for recording node states. Key features include:

- Retrieving the last state and history
- Supports state rollback
- Allows data modification
- Uses thread_id and thread_ts to uniquely locate archives

```
graph.get_state(config)  # Get the last saved state
graph.get_state_history(config)  # Get the list of all saved states
```

YHK

Best Practice Suggestions

- Choose the appropriate data processing method according to needs, considering using induction functions to handle cumulative data.
- Pay attention to the hierarchy and connection of nodes when designing graph structures to achieve the desired execution flow.
- Make reasonable use of the checkpoint mechanism, but be aware of storage overhead.
- When dealing with complex states, consider using TypedDict and Annotated to enhance type hints and data processing logic.

YHK

Considerations

- The default data overwrite behavior may lead to unexpected results, so handle state updates with care.
- In parallel execution of multiple nodes, be aware of the impact of uncertain execution order.
- Consider performance impacts when using the checkpoint mechanism, especially when dealing with large amounts of data or frequent archiving.
- Although induction functions provide convenience, they may increase complexity in certain special operations, requiring a trade-off in use.

YHK

Streaming Response in LangGraph

- Different from traditional LLM word-by-word output
- Outputs node data state each time for granular control
- Values mode returns complete graph state (total)
- Updates mode returns state changes only (incremental)
- Compiled graph is essentially a Runnable component
- Multiple streaming modes for different use cases
- Enhanced user experience through real-time feedback
- Future improvements for node-level streaming

YHK

Streaming Modes Implementation

- Values mode: Complete state after each node
- Updates mode: Incremental changes as dictionary
- Dictionary keys represent node names
- Dictionary values contain state updates
- Choose mode based on application requirements

```
# Values mode: Returns complete state values
inputs = {"messages": [("human", "What are the top 3 results of the 2024
    Beijing Half Marathon?")]}

for chunk in agent.stream(inputs, stream_mode="values"):
    print(chunk["messages"][-1].pretty_print())

# Updates mode: Returns state updates only
for chunk in agent.stream(inputs, stream_mode="updates"):
    print(chunk)

```

YHK

Multi Agents in LangGraph

(Ref: LangGraph Crash Course - Harish Neel)

YHK

Multi-Agent Systems: Problems and Solutions

- Agent is a system using LLM to control application flow
- Complex systems become harder to manage and scale over time
- Problem: Agents have too many tools, make poor decisions
- Problem: Context grows too complex for single agent tracking
- Problem: Need for multiple specialization areas (planning, research, math)
- Solution: Break application into smaller independent agents
- Independent agents can be simple prompt+LLM or complex React agents
- Primary benefit: Modularity makes development, testing, maintenance easier
- Primary benefit: Specialization creates expert domain-focused agents
- Primary benefit: Control explicit agent communication vs function calling

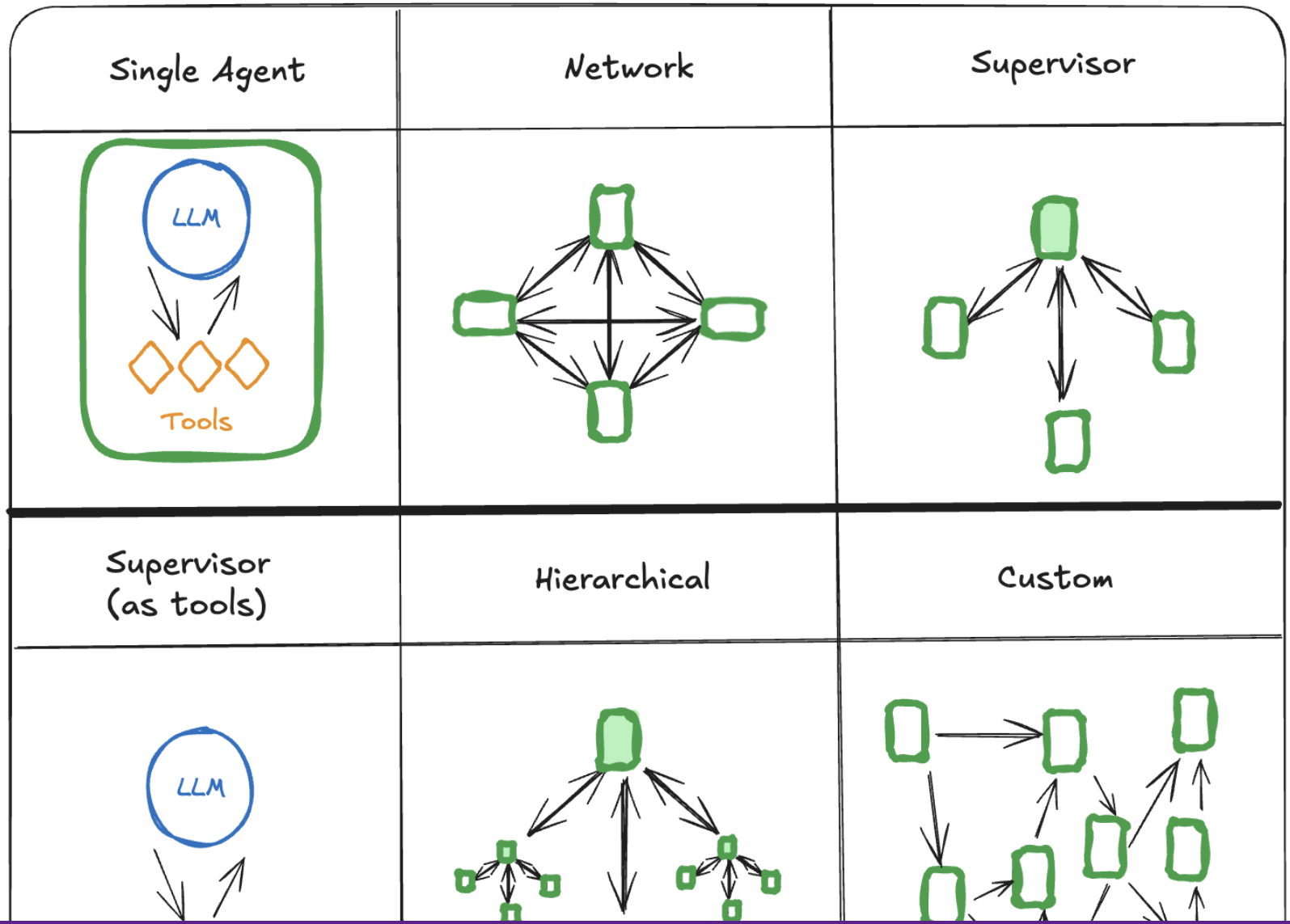
YHK

Multi-Agent Architecture Types

- Single Agent: Basic LLM with tools attachment
- Network: Four agents communicating with each other directly
- Network: Agents solve task parts, hand control to next agent
- Supervisor: One supervisor orchestrates team of specialized employees
- Supervisor: Coding agent, research agent, validation agent examples
- Supervisor: Agents don't communicate directly, only through supervisor
- Supervisor as Tools: Same as supervisor but agents provided as LLM tools
- Hierarchical: Extension of supervisor with multiple supervisor levels
- Hierarchical: Similar to big company organizational structure
- Custom: No clear definitions, conditional flow based on agent outputs

YHK

Multi-Agent Architecture Types



Subgraphs: Foundation for Multi-Agent Systems

- Subgraphs build complex systems with multiple graph components
- Common use case: Building multi-agent systems architecture
- Key question: How parent graph and subgraph communicate
- Key question: How to pass state between graphs during execution
- Scenario 1: Parent and subgraph share same schema keys
- Scenario 1: Add node with compiled subgraph directly
- Scenario 2: Parent and subgraph have different schemas
- Scenario 2: Add node function that invokes subgraph with transformation
- Transformation useful when state schemas differ between graphs
- Foundation enables building complex multi-agent architectures

YHK

Implementing Subgraphs: Code Examples

- Case 1: Shared schema - direct embedding approach
- Case 2: Different schemas - intermediate transformation node required
- Child graph: Simple LLM with Tavily search tool integration
- Parent graph: Embeds child graph as specialized search agent
- Shared schema: Compile subgraph, add as parent graph node
- Different schema: Extract query, transform to messages format
- Different schema: Invoke subgraph, extract response from result
- State transformation: Convert between parent and child schemas
- Example: Weather query routing through search agent subgraph
- Pattern enables modular, scalable multi-agent system construction

YHK

Implementing Subgraphs: Code Examples

Parent and subgraph have shared state keys in their state schemas. In this case, you can include the subgraph as a node in the parent graph

```python
from langgraph.graph import StateGraph, MessagesState, START

# Subgraph

def call_model(state: MessagesState):
    response = model.invoke(state["messages"])
    return {"messages": response}

subgraph_builder = StateGraph(State)
subgraph_builder.add_node(call_model)
...
subgraph = subgraph_builder.compile()

# Parent graph

builder = StateGraph(State)
builder.add_node("subgraph_node", subgraph)
builder.add_edge(START, "subgraph_node")
graph = builder.compile()
...
graph.invoke({"messages": [{"role": "user", "content": "hi!"}]})
```

YHK

Implementing Subgraphs: Code Examples

Parent graph and subgraph have different schemas (no shared state keys in their state schemas). In this case, you have to call the subgraph from inside a node in the parent graph

```python
from typing_extensions import TypedDict, Annotated
from langchain_core.messages import AnyMessage
from langgraph.graph import StateGraph, MessagesState, START
from langgraph.graph.message import add_messages

class SubgraphMessagesState(TypedDict):
    subgraph_messages: Annotated[list[AnyMessage], add_messages]

# Subgraph

def call_model(state: SubgraphMessagesState):
    response = model.invoke(state["subgraph_messages"])
    return {"subgraph_messages": response}

subgraph_builder = StateGraph(SubgraphMessagesState)
subgraph_builder.add_node("call_model_from_subgraph", call_model)
subgraph_builder.add_edge(START, "call_model_from_subgraph")
...
subgraph = subgraph_builder.compile()

# Parent graph

def call_subgraph(state: MessagesState):
    response = subgraph.invoke({"subgraph_messages": state["messages"]})
    return {"messages": response["subgraph_messages"]}

builder = StateGraph(State)
builder.add_node("subgraph_node", call_subgraph)
builder.add_edge(START, "subgraph_node")
```

YHK

Implementation

Node Implementation Examples

Example: Node with RAG

```python
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings, ChatOpenAI

# Initialize RAG components
vectorstore = FAISS.load_local("docs_index", OpenAIEmbeddings())
llm = ChatOpenAI(model="gpt-4")

def rag_node(state: State) -> State:
    """Node that performs RAG operation"""

    # 1. Retrieve relevant documents
    query = state.messages[-1]["content"]
    docs = vectorstore.similarity_search(query, k=3)
    context = "\n".join([doc.page_content for doc in docs])

    # 2. Generate answer with context
    prompt = f"""Context: {context}

Question: {query}

Answer based on the context provided:"""

    response = llm.invoke(prompt)

    # 3. Update state
    return State(
        messages=state.messages + [{"role": "assistant", "content": response.content}],
        retrieved_docs=docs
    )

```

Example: Node with API Call

Any REST API, database query, or external service can be called from nodes

```python
import requests

def weather_api_node(state: State) -> State:
    """Node that calls external weather API"""

    # Extract location from state
    location = state.location

    # Call external API
    api_key = "your_api_key"
    url = f"https://api.openweathermap.org/data/2.5/weather?q={location}&appid={api_key}"
    response = requests.get(url)
    weather_data = response.json()

    # Process and update state
    weather_info = f"Temperature: {weather_data['main']['temp']}deg C, "
    weather_info += f"Conditions: {weather_data['weather'][0]['description']}"

    return State(
        location=state.location,
        weather=weather_info,
        messages=state.messages + [{"role": "system", "content": weather_info}]
    )
```

YHK

Example: Node as Tool-Calling Agent

- Node delegates to inner agent for complex tool-using tasks
- Outer graph manages high-level workflow

```python
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain.tools import Tool

# Define tools
search_tool = Tool(name="search", func=tavily_search, description="Search the web")
calc_tool = Tool(name="calculator", func=calculator, description="Perform calculations")

# Create agent with tools
tools = [search_tool, calc_tool]
agent = create_tool_calling_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools)

def research_node(state: State) -> State:
    """Node that is itself an agent with tools"""

    # This node runs a full agent loop internally
    query = state.current_task
    result = agent_executor.invoke({"input": query})

    return State(
        current_task=state.current_task,
        research_results=result["output"],
        messages=state.messages + [{"role": "researcher", "content": result["output"]}]
    )

```

End-to-End example

Implementing a Complete Tool-Calling Agent

- Understand user input
- Select the appropriate tool
- Execute tool calls
- Generate the final response

YHK

Define State

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

from langchain_core.language_models import ChatOpenAI

class Tool(BaseModel):
    name: str
    description: str
    func: callable

class AgentState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    thought: str = ""
    selected_tool: Optional[str] = None
    tool_input: str = ""
    tool_output: str = ""
    final_answer: str = ""
    status: str = "STARTING"

```

Define Tools

```

from langchain.tools import BaseTool
from langchain.tools.calculator import CalculatorTool
from langchain.tools.wikipedia import WikipediaQueryRun
from langchain_core.language_models import ChatOpenAI

# Define available tools
tools = [
    Tool(
        name="calculator",
        description="Used for performing mathematical calculations",
        func=CalculatorTool()
    ),
    Tool(
        name="wikipedia",
        description="Used for querying Wikipedia information",
        func=WikipediaQueryRun()
    )
]

```

Implement Core Nodes

```python
async def think(state: AgentState) -> AgentState:
    """Think about the next action"""
    prompt = f"""
    Based on user input and current conversation history, think about the next action.
    User input: {state.current_input}
    Available tools: {[t.name + ': ' + t.description for t in tools]}
    Decide:
    1. Whether a tool is needed
    2. If needed, which tool to use
    3. What parameters to call the tool with
    Return in JSON format: {{"thought": "thought process", "need_tool": true/false, "tool": "tool name", "tool_input":
        "parameters"}}
    """
    llm = ChatOpenAI(temperature=0)
    response = await llm.ainvoke(prompt)
    result = json.loads(response)
    return AgentState(
        **state.dict(),
        thought=result["thought"],
        selected_tool=result.get("tool"),
        tool_input=result.get("tool_input"),
        status="NEED_TOOL" if result["need_tool"] else "GENERATE_RESPONSE"
    )
```

Implement Core Nodes

```python
async def execute_tool(state: AgentState) -> AgentState:
    """Execute tool call"""
    tool = next((t for t in tools if t.name == state.selected_tool), None)
    if not tool:
        return AgentState(
            **state.dict(),
            status="ERROR",
            thought="Selected tool not found"
        )
    try:
        result = await tool.func.ainvoke(state.tool_input)
        return AgentState(
            **state.dict(),
            tool_output=str(result),
            status="GENERATE_RESPONSE"
        )
    except Exception as e:
        return AgentState(
            **state.dict(),
            status="ERROR",
            thought=f"Tool execution failed: {str(e)}"
        )
```

Implement Core Nodes

```python
async def generate_response(state: AgentState) -> AgentState:
    """Generate the final response"""
    prompt = f"""
    Generate a response to the user based on the following information:
    User input: {state.current_input}
    Thought process: {state.thought}
    Tool output: {state.tool_output}
    Please generate a clear and helpful response.
    """
    llm = ChatOpenAI(temperature=0.7)
    response = await llm.ainvoke(prompt)
    return AgentState(
        **state.dict(),
        final_answer=response,
        status="SUCCESS"
    )
```

Build the Complete Workflow

```
# Create graph structure
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("think", think)
workflow.add_node("execute_tool", execute_tool)
workflow.add_node("generate_response", generate_response)

# Add edges
workflow.add_edge("think", "execute_tool", condition=lambda s: s.status == "NEED_TOOL")
workflow.add_edge("execute_tool", "generate_response", condition=lambda s: s.status == "GENERATE_RESPONSE")
workflow.add_edge("generate_response", "think", condition=lambda s: s.status == "SUCCESS")
```

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://medium.com/@yogeshharibhaukulkarni)

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

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

YHK