

BUILDING ADVANCED AI AGENTS FOR WORKFLOW AUTOMATION WITH LANGGRAPH

Yogesh Haribhau Kulkarni



Outline

1 OVERVIEW

2 CONCLUSIONS

About Me

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.

Introduction to LangGraph

(Ref: LangGraph Crash Course - Harish Neel)

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 → intelligent chains → 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

Levels of Autonomy/Agency/Freedom: Overview

- ▶ Think of autonomy as a spectrum from rigid to flexible
- ▶ Each level trades off control for capability
- ▶ Understanding these levels helps you choose the right tool
- ▶ **The Spectrum:**
 1. Code: No autonomy, 100% deterministic
 2. LLM Call: Single intelligent decision
 3. Chains: Sequential reasoning
 4. Routers: Conditional branching
 5. State: Workflow modeling with Cycles
 6. Agents: Autonomous decision-making with loops
- ▶ LangGraph enables 'State' level

Level 1-2: Code and Single LLM Calls

► Level 1: Pure Code:

- Example: `if temperature > 30: return "It's hot"`
- No autonomy, 100% deterministic, predictable
- Problem: Must anticipate every scenario
- Use case: Simple rule-based logic

► Level 2 - Single LLM Call:

- Example: "Summarize this article" → LLM → Summary
- One input one output. One atomic task, limited reasoning
- Problem: Trying to do everything in one shot. Cannot break down complex tasks
- Use case: Text generation, classification, extraction

► Limitation: Both lack multi-step reasoning capability

Level 3: Chains: Sequential Reasoning

- ▶ **What:** Unidirectional sequence of LLM operations, so multiple experts one after another. Breaking down task is possible now.
- ▶ **Example: Customer Support:**
 - ▶ Step 1: Classify intent (complaint/question/request)
 - ▶ Step 2: Extract key information (order ID, issue)
 - ▶ Step 3: Generate appropriate response
- ▶ **Flow:** $A \rightarrow B \rightarrow C \rightarrow \text{End}$ (no branching, no loops)
- ▶ **Advantages:** Predictable, reliable, easy to debug
- ▶ **Limitations:**
 - ▶ Cannot adapt to different scenarios
 - ▶ No conditional logic or branching
 - ▶ Fixed execution path regardless of context
- ▶ Like an assembly line, efficient but inflexible

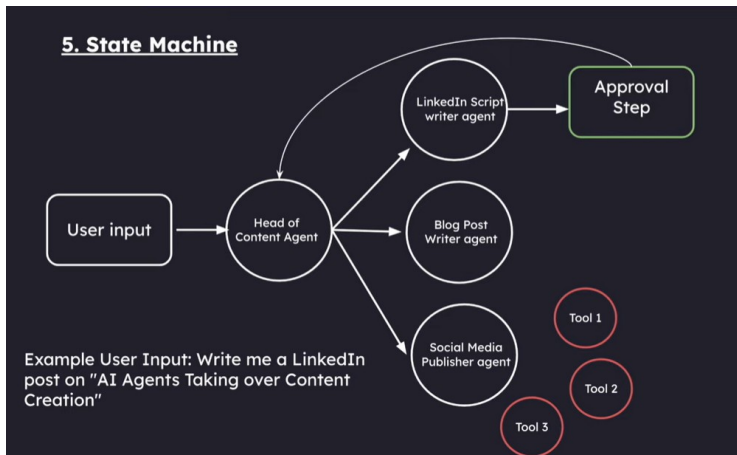
Level 4: Routers: Conditional Branching

- ▶ **What:** Chains with decision points, but still unidirectional
- ▶ **Example: Document Processing:**
 - ▶ Analyze document type, LLM/ML, decision making (invoice/contract/email)
 - ▶ IF invoice → Extract line items → Calculate total
 - ▶ IF contract → Extract parties → Identify key terms
 - ▶ IF email → Classify urgency → Route to handler
- ▶ **Flow:** Decision node routes to specialized chains
- ▶ **Advantages:** Handles different scenarios, specialized processing
- ▶ **Limitations:**
 - ▶ Still no loops or backward flow
 - ▶ Cannot learn from mistake, cannot revisit back
 - ▶ One-way path through the system
- ▶ Like a train switching tracks - more flexible but still one direction

Level 5: State: Flow with Loops

- ▶ **What:** State machines are Routers who can branch and loop.
- ▶ **Example: Content Creator:**
 - ▶ Given a theme to write on
 - ▶ HEAD can sending it for LinkedIn article writing,
 - ▶ Once done, we can have Human-approval, if not approved go back to the HEAD. Revise, iterate till approved.
 - ▶ Once approved send to other forms such as Blog post, Twitter announcement post etc.
- ▶ **Key Features:**
 - ▶ Can have human-in-loop for approval
 - ▶ Multi-Agent systems
 - ▶ Advanced Memory Management
 - ▶ Can go back in history and explore better/alternate paths
- ▶ **Challenge:** Flow has to be prescribed, not full autonomy.
- ▶ **This is LangGraph**

Level 5: State: Flow with Loops

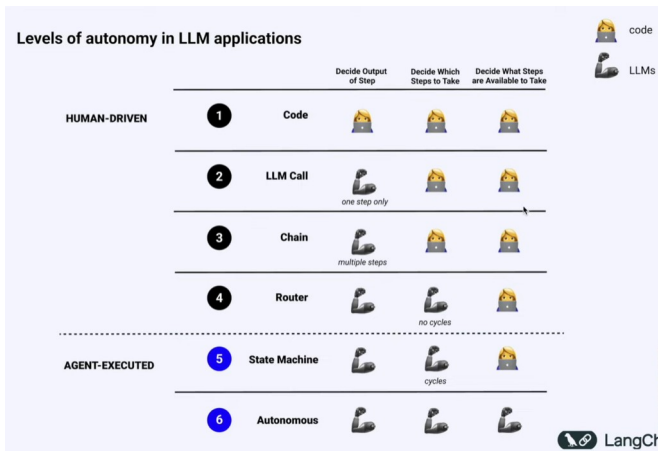


(Ref: LangGraph Crash Course - Harish Neel)

Level 6: Agents: Autonomous with Loops

- ▶ **What:** State machines with independent thinking, can loop and branch
- ▶ **Example - Research Assistant:**
 - ▶ User: "Analyze SpaceX's latest launch success rate"
 - ▶ Agent thinks: "I need current data and historical context"
 - ▶ Action 1: Search web for latest SpaceX launches
 - ▶ Observation: Found data but it's incomplete
 - ▶ Action 2: Search for historical launch records
 - ▶ Observation: Now have full picture
 - ▶ Final answer: Synthesize findings into analysis
- ▶ **Key Features:** Think → Act → Observe → Decide → Repeat
- ▶ **Challenge:** Can loop infinitely if not controlled

Levels of Autonomy in LLM Applications



(Ref: LangGraph Crash Course - Harish Neel)

LLM (What AI decides) vs Code (What we prescribe/program)

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.

Introduction to LangGraph

What is LangGraph?



LangGraph

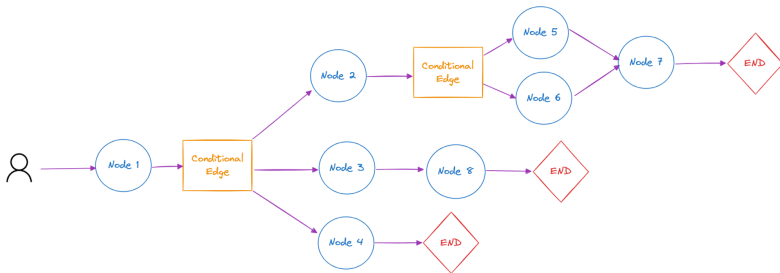
- ▶ LangGraph is an extension of the popular LangChain library.
- ▶ Allows you to create AI applications that can perform multiple steps, make decisions, and maintain information across those steps.
- ▶ Think of it like building a flowchart for your AI to follow.

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

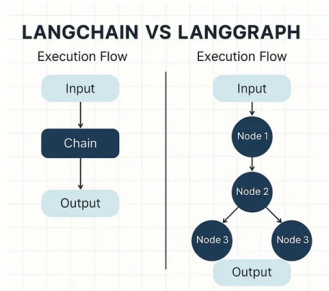
How is LangGraph Different?

- ▶ 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 - $A \rightarrow B \rightarrow C \rightarrow \text{End}$
- ▶ **Real-World Problems:** Require branching, loops, and conditional paths
- ▶ **Graph Advantages:**
 - ▶ Conditional branching: "If quality \geq 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

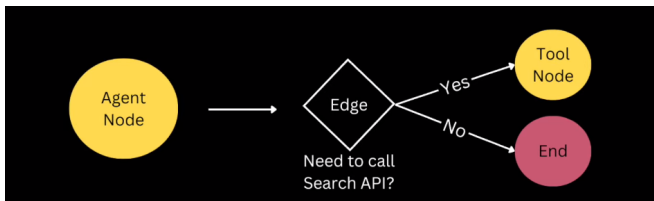
Agentic Patterns: Autonomous vs Workflow Automation

- ▶ **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

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



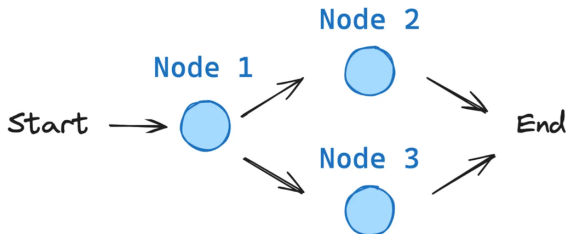
(Ref: Introduction to LangGraph — Building an AI Generated Podcast - Prompt Circle AI)

-
- Diagram illustrating the stateful nature of the LLM agent:
- Initial State:** A user icon (person) sends a message: `(messages:[{"type":"human","content":"what are solar panels"}])`.
 - Agent Node (Green Circle):** Labeled **Node**. Contains the text: **Agent**
Interacts with customer.
Obtains monthly energy cost.
 - Edge (Yellow Oval):** Labeled **Edge**. Contains the text: **Where to go**.
 - Tool Node (Green Circle):** Labeled **Node**. Contains the text: **Tool**
energy_cost_estimate.py.
 - END Node (Red Oval):** Labeled **END**.
- Transitions are labeled **State**:
- From the initial state to the **Agent Node**.
 - From the **Agent Node** to the **Edge**.
 - From the **Edge** to the **Tool Node**.
 - From the **Tool Node** to the **END Node**.
 - A curved arrow labeled **State** points from the **Tool Node** back to the **Agent Node**, indicating a feedback loop.

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)

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

What Can Nodes Do? - Part 1

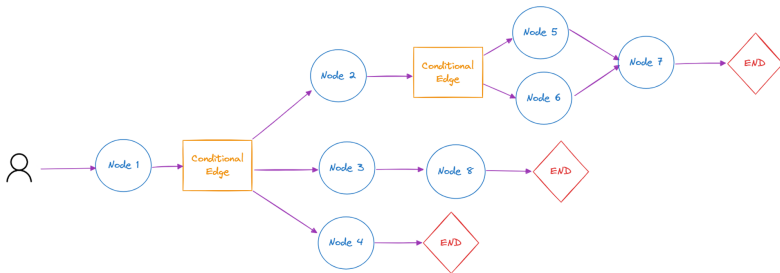
- ▶ **Nodes are Extremely Flexible** - any Python function
- ▶ **1. Simple LLM Calls:**
 - ▶ Generate text, classify, extract information
 - ▶ Example: `llm.invoke("Summarize this text")`
- ▶ **2. RAG Operations:**
 - ▶ Retrieve documents from vector database
 - ▶ Rerank results
 - ▶ Generate answers 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

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

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)

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

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

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

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 response drafter
 4. Draft response for legitimate emails
 5. Send notification
- ▶ Demonstrates conditional routing and structured workflow
- ▶ Shows how graphs handle if/else logic naturally

Installation

LangGraph requires Python 3.8 or later.

```
1 pip install -U langgraph
2
3
4 python -c "import langgraph; print(langgraph.__version__)"
```

Step 1: Define State

- ▶ State holds all information passed between nodes
- ▶ Type hints ensure clarity and validation
- ▶ DO NOTE, in the following functions, the incoming state object is not updated with newer values of the field, but a new State object is returned.

```
1 from typing import List, Dict
2 from pydantic import BaseModel
3
4 class EmailState(BaseModel):
5     email_content: str = ""
6     is_spam: bool = False
7     category: str = ""
8     draft_response: str = ""
9     messages: List[Dict[str, str]] = []
10
11
12 # def classify_email(state: EmailState) -> EmailState:
13 #     # WRONG - Mutating incoming state
14 #     state.is_spam = True # DON'T DO THIS
15 #     state.messages.append(...) # DON'T DO THIS
16 #     return state
17
```

Step 2: Create Node Functions

```
import os
2 from langchain_groq import ChatGroq

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

6 def read_email(state: EmailState) -> EmailState:
    return state

8

def classify_email(state: EmailState) -> EmailState:
10     prompt = f"Classify this email as spam or
        legitimate:\n{state.email_content}\nRespond with only 'spam' or
        'legitimate'."
    response = llm.invoke(prompt)
12     is_spam = "spam" in response.content.lower()
    return EmailState(
14         email_content=state.email_content,
        is_spam=is_spam,
16         category=state.category,
        draft_response=state.draft_response,
18         messages=state.messages + [{"role": "classifier", "content":
            response.content}])

20
```

Step 2: Create Node Functions

```
1 def handle_spam(state: EmailState) -> EmailState:
    new_messages = state.messages + [{"role": "system", "content": "Email
    marked as spam"}]
3     return EmailState(
        email_content=state.email_content,
5         is_spam=state.is_spam,
        category=state.category,
7         draft_response=state.draft_response,
        messages=new_messages
9     )

11 def draft_response(state: EmailState) -> EmailState:
    prompt = f"Draft a professional response to:\n{state.email_content}"
13     response = llm.invoke(prompt)
    return EmailState(
15         email_content=state.email_content,
        is_spam=state.is_spam,
17         category=state.category,
        draft_response=response.content,
19         messages=state.messages + [{"role": "drafter", "content":
        response.content}]
    )
21
```


Step 3: Define Routing Logic

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

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

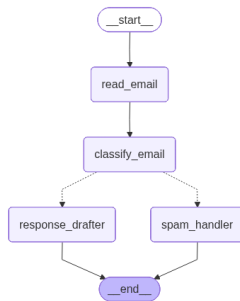
Step 4: Build the Graph

```
1 from langgraph.graph import StateGraph, START, END
2
3 workflow = StateGraph(EmailState) # Create graph
4
5 workflow.add_node("read_email", read_email)
6 workflow.add_node("classify_email", classify_email)
7 workflow.add_node("spam_handler", handle_spam)
8 workflow.add_node("response_drafter", draft_response)
9
10 workflow.add_edge(START, "read_email")
11 workflow.add_edge("read_email", "classify_email")
12
13 def route_email(state: EmailState) -> str:
14     if state.is_spam:
15         return "spam_handler" # Exact node name
16     else:
17         return "response_drafter" # Exact node name
18
19 workflow.add_conditional_edges("classify_email", route_email)
20 workflow.add_edge("spam_handler", END)
21 workflow.add_edge("response_drafter", END)
22
23 app = workflow.compile()
24
```

Step 5: Visualize the Graph

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

```
1 # Visualize and save the graph
  png_graph = app.get_graph().draw_mermaid_png()
3 with open("langgraph_email_workflow_graph.png", "wb")
  as f:
    f.write(png_graph)
5 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

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

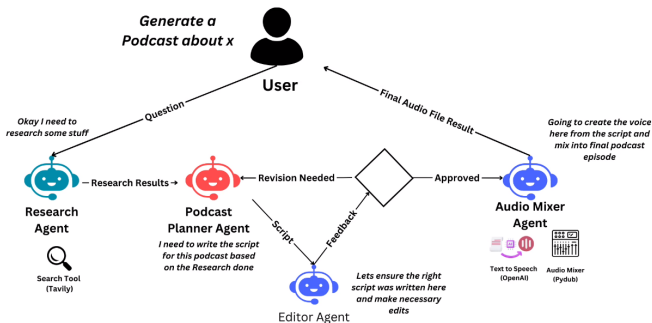
9 # Test with spam email
  spam_state = EmailState(
11     email_content="URGENT! You've won $1M! Click here now!!!"
  )
13 spam_result = app.invoke(spam_state)
  print(f"Is Spam: {spam_result['is_spam']}")
15
```

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

Real-World Applications

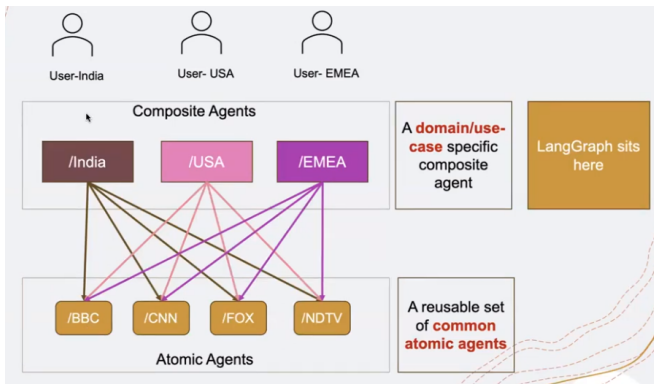
Example: Podcast Generator



(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)

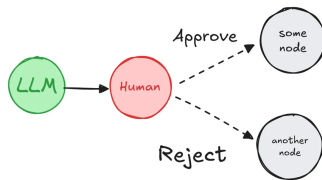
Best Practices

- ▶ **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

Advanced Concepts in LangGraph

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

Human-in-the-Loop: Code Example

```
2 from langgraph.graph import StateGraph
4 # Create graph with checkpointing (required for interrupts)
4 memory = MemorySaver()
   workflow = StateGraph(State)
6
   workflow.add_node("draft_email", draft_email_node)
8 workflow.add_node("send_email", send_email_node)
   workflow.add_edge("draft_email", "send_email")
10 # Compile with interrupt BEFORE send_email node
   app = workflow.compile(
12     checkpointer=memory,
       interrupt_before=["send_email"]) # Pause here for human approval
14
```

Human-in-the-Loop: Code Example

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

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

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

26 # To modify state and resume:
   updated_state = result.copy()
28 updated_state.draft = "Modified email content"
   app.invoke(updated_state, config)
30
```

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

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..." → "searching..." → "responding..."

Streaming Example

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

```
# Instead of invoke(), use stream()
2 for chunk in app.stream(initial_state, config):
    # chunk contains state updates after each node
    4     print(f"Node: {chunk['node']}")
    print(f"State: {chunk['state']}")
    6
# For token-by-token LLM streaming
8 async for event in app.astream_events(initial_state, config):
    if event["event"] == "on_chat_model_stream":
        10         # Stream each token as LLM generates
        print(event["data"]["chunk"], end="", flush=True)
        12
```

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

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

```
1 # Parallel node execution
2 workflow.add_node("search_web", search_node)
3 workflow.add_node("search_db", database_node)
4 workflow.add_node("search_docs", documents_node)
5 workflow.add_node("aggregate", aggregate_results)
6 # Fan-out: one node to many
7 workflow.add_edge("start", "search_web")
8 workflow.add_edge("start", "search_db")
9 workflow.add_edge("start", "search_docs")
10 # Fan-in: many nodes to one
11 workflow.add_edge("search_web", "aggregate")
12 workflow.add_edge("search_db", "aggregate")
13 workflow.add_edge("search_docs", "aggregate")
```

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

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

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

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

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

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

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

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

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/>)