🔥 2025 EDITION

Top 20 Interview Questions

on LangGraph

Master graph-based AI agents and ace your next interview

📊 State Management

🔄 Checkpointing

🤖 Multi-Agent

👤 Human-in-Loop

Naved Khan
Gen AI Engineer

+ Follow

Q1 / 20

EASY

What is LangGraph and how does it differ from LangChain?

ANSWER

LangGraph is a framework for building stateful, multi-actor applications with LLMs. It represents workflows as **graphs** where nodes are functions and edges define transitions.

LangChain	LangGraph
Sequential chains (DAGs)	Graph-based (cycles OK)
No built-in cycles	Built-in state management
Simple workflows	Complex workflows

KEY DIFFERENCES

→ LangGraph supports **cycles** (retry loops, iterative agents)

→ Built-in **checkpointing** for persistence

→ Native **multi-agent** coordination

Q2 / 20

EASY

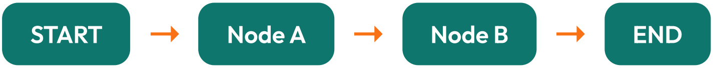
What are the core concepts: Nodes, Edges, and State?

💡 **ANSWER**

LangGraph workflows are defined by three core building blocks that work together to create complex agent flows.

BUILDING BLOCKS

- 📦 **Nodes:** Python functions that process and return state updates
- 🔗 **Edges:** Define transitions between nodes (normal or conditional)
- 📊 **State:** TypedDict that flows through graph, accumulating results


START → Node A → Node B → END

```
graph = StateGraph(State)
graph.add_node("agent", agent_fn)
graph.add_edge(START, "agent")
graph.add_edge("agent", END)
```

Q3 / 20

MEDIUM

How does state management work in LangGraph?

ANSWER

State is a shared data structure (TypedDict) that persists across nodes. Each node receives current state and returns updates that get **merged** back.

```
from typing import TypedDict

class State(TypedDict):
  messages: list
  current_step: str
  results: dict
```

STATE FLOW

1. Initial state provided at `invoke()`
2. Each node receives current state
3. Node returns partial updates
4. Updates merged into state → next node

INTERVIEW TIP

Q4 / 20

HARD

What is TypedDict and Annotated in state definition?

💡 ANSWER

TypedDict defines the state schema. **Annotated** with a **reducer function** specifies how state updates are merged (append vs replace).

```
from typing import Annotated
import operator

class State(TypedDict):
  # Appends new messages to list
  messages: Annotated[list, operator.add]
  # Replaces value (default)
  current_step: str
```

REDUCER FUNCTIONS

- → `operator.add` : Append lists together
- → `add_messages` : Smart message merging (deduplication)
- → No annotation: Replace value entirely

⚠ **COMMON TRAP**

Q5 / 20

MEDIUM

What is MessagesState and add_messages?

ANSWER

MessagesState is a pre-built state with a `messages` list using **add_messages** reducer. It handles message deduplication and proper merging.

```
from langgraph.graph import MessagesState

# Equivalent to:
from langgraph.graph.message import add_messages
class State(TypedDict):
  messages: Annotated[list, add_messages]
```

ADD_MESSAGES FEATURES

- ✓ Appends new messages to existing list
- ✓ Deduplicates by message ID
- ✓ Updates existing messages if same ID
- ✓ Handles BaseMessage objects properly

BEST PRACTICE

Use MessagesState for chatbots - it's optimized for conversation flows!

Q6 / 20

MEDIUM

What are conditional edges and how to implement them?

ANSWER

Conditional edges route to different nodes based on state. A **routing function** examines state and returns the next node name.

```
def should_continue(state):
  if state["done"]:
    return "end"
  return "continue"

graph.add_conditional_edges(
  "agent",
  should_continue,
  {"end": END, "continue": "tools"})
```

USE CASES

→ Tool calling decisions (call tool or respond)

→ Loop until condition (retry logic)

→ Route by classification result

Q7 / 20

MEDIUM

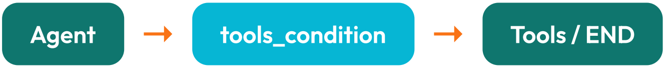
What is tools_condition in LangGraph?

💡 **ANSWER**

tools_condition is a prebuilt routing function that checks if the last message has **tool calls**. Routes to "tools" node or END.

```
from langgraph.prebuilt import tools_condition

graph.add_conditional_edges(
  "chatbot",
  tools_condition
)
# Routes to "tools" or END automatically
```



HOW IT WORKS

- ✓ Checks `tool_calls` in last AI message
- ✓ Returns "tools" if tool calls exist
- ✓ Returns END if no tool calls (final response)

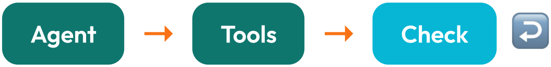
Q8 / 20

HARD

How to build an agent loop that retries until success?

💡 ANSWER

LangGraph enables **cycles** - edges that loop back to previous nodes. The agent continues until a condition routes to END.



Agent → Tools → Check ↩

```
graph.add_edge("tools", "agent")
# Creates cycle: agent → tools → agent

graph.add_conditional_edges(
  "agent", tools_condition)
```

AGENT LOOP PATTERN

1. Agent decides: call tool or respond
2. If tool → execute → return to agent
3. Repeat until no more tool calls → END

Naved Khan
Gen AI Engineer

Q9 / 20

MEDIUM

What is checkpointing and why is it important?

💡 ANSWER

Checkpointing saves graph state at each step, enabling **persistence**, **resumption** after failures, and **time-travel debugging**.

```python
from langgraph.checkpoint.memory import InMemorySaver

memory = InMemorySaver()
graph = builder.compile(checkpointer=memory)
```

CHECKPOINTING ENABLES

- 💾 **Persistence:** Save conversation state across sessions
- 🔄 **Resumption:** Continue from last point after failure
- ⏰ **Time-travel:** Go back to any previous state
- 👤 **Human-in-loop:** Pause for approval

Q10 / 20

MEDIUM

What are the different checkpointer options?

💡 **ANSWER**

LangGraph provides multiple checkpointers for different persistence needs - from development to production scale.

CHECKPOINTER TYPES

- 🧪 **InMemorySaver:** Development & testing (non-persistent)
- 📁 **SqliteSaver:** Local file persistence
- 🐘 **PostgresSaver:** Production database persistence
- 🌿 **MongoDBSaver:** Document-based persistence

```
# Development
from langgraph.checkpoint.memory import InMemorySaver
memory = InMemorySaver()

# Production
from langgraph.checkpoint.postgres import PostgresSaver
checkpointer = PostgresSaver.from_conn_string(DB_URI)
```

Q11 / 20

MEDIUM

How does thread_id work for persistence?

ANSWER

thread_id is a unique identifier for a conversation session. Each thread maintains its own state history, enabling multi-user support.

```
config = {
  "configurable": {
    "thread_id": "user_123"
  }
}

# First message
graph.invoke(input_1, config)

# Same thread - has memory!
graph.invoke(input_2, config)
```

THREAD ID USE CASES

- → User-specific conversation history
- → Multiple parallel conversations
- → Resume sessions across requests

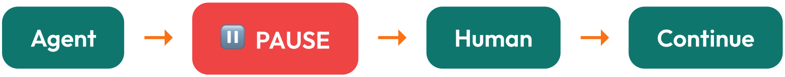
Q12 / 20

HARD

What is human-in-the-loop in LangGraph?

💡 ANSWER

Human-in-the-loop pauses graph execution at specific points for human review, approval, or input before continuing.



USE CASES

- ✅ Approve before executing sensitive actions
- ✏️ Edit agent's proposed response
- 🔧 Correct mistakes before they propagate
- ⚠️ Quality control checkpoints

💡 REQUIRES CHECKPOINTING

Human-in-the-loop needs a checkpointer to save state while waiting for human input!

Q13 / 20

HARD

How to implement interrupt points?

💡 **ANSWER**

Use **interrupt_before** or **interrupt_after** at compile time to specify which nodes trigger a pause.

```python
graph = builder.compile(
  checkpointer=memory,
  interrupt_before=["tools"]
)

# Resume after human approval
graph.invoke(None, config)
```

interrupt_before

Pause BEFORE node executes
Review what will happen

interrupt_after

Pause AFTER node executes
Review results before next

⚡ **RESUME EXECUTION**

Call `invoke(None, config)` to continue from interrupt point!

Q14 / 20

HARD

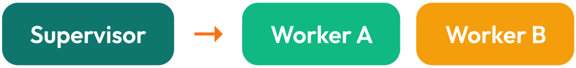
How to build multi-agent systems in LangGraph?

💡 **ANSWER**

Multi-agent systems use multiple agent nodes that coordinate through shared state. Common patterns include **supervisor**, **collaborative**, and **debate**.

MULTI-AGENT PATTERNS

- 👔 **Supervisor:** One agent delegates to worker agents
- 🤝 **Collaborative:** Agents build on each other's work
- ⚔️ **Debate:** Agents argue positions to refine output



```
graph.add_node("supervisor", supervisor_fn)
graph.add_node("researcher", researcher_fn)
graph.add_node("writer", writer_fn)
```

Q15 / 20

HARD

What are subgraphs and when to use them?

💡 **ANSWER**

Subgraphs are nested graphs that encapsulate reusable workflows. Use a compiled graph as a node in a parent graph.

```
# Create subgraph
subgraph = subgraph_builder.compile()

# Use as node in parent
main_graph.add_node("research", subgraph)
main_graph.add_edge("research", "write")
```

BENEFITS OF SUBGRAPHS

- ♻️ **Reusability:** Same subgraph in multiple workflows
- 🧪 **Testability:** Test subgraphs in isolation
- 📦 **Organization:** Modular, maintainable code
- 🔒 **Encapsulation:** Hide internal complexity

💡 CHECKPOINTER PROPAGATION

Parent's checkpointer automatically propagates to subgraphs!

Q16 / 20

MEDIUM

How does streaming work in LangGraph?

💡 **ANSWER**

LangGraph supports multiple **streaming modes** to output results as they're generated, improving perceived latency.

```python
for chunk in graph.stream(
  inputs,
  config,
  stream_mode="values"
):
  print(chunk)
```

STREAM MODES

→ **"values"** : Full state after each node

→ **"updates"** : Only state changes per node

→ **"messages"** : Stream individual tokens

→ **"debug"** : Detailed execution info

💡 **BEST PRACTICE**

Use "updates" for efficient streaming, "values" for debugging full state

Q17 / 20

HARD

What is time-travel debugging in LangGraph?

💡 **ANSWER**

Time-travel lets you go back to any previous checkpoint, inspect or modify state, and re-run from that point.

```
# Get all checkpoints
states = list(graph.get_state_history(config))

# Go back to specific checkpoint
to_replay = states[2]
# Resume from checkpoint
graph.invoke(None, to_replay.config)
```

TIME-TRAVEL USE CASES

- 🔍 Debug issues by inspecting past states
- ↩️ Undo and retry with different input
- 🔀 Branch from past state for "what-if" analysis

Q18 / 20

MEDIUM

What is ToolNode in LangGraph?

💡 **ANSWER**

ToolNode is a prebuilt node that executes tool calls from the last AI message. It handles tool execution and returns results.

```
from langgraph.prebuilt import ToolNode

tools = [search_tool, calc_tool]
tool_node = ToolNode(tools=tools)

graph.add_node("tools", tool_node)
```

TOOLNODE FEATURES

✓ Extracts tool_calls from AI message

✓ Executes matching tool with arguments

✓ Returns ToolMessage with results

✓ Handles multiple tool calls in parallel

🔧 **COMPLETE AGENT SETUP**

ToolNode + tools_condition = Complete ReAct agent pattern!

Q19 / 20

EASY

LangGraph vs LangChain - when to use which?

💡 **ANSWER**

Choose based on workflow complexity. **LangChain** for simple chains, **LangGraph** for stateful, complex agent workflows.

Use LangChain

Simple A→B→C flows
Basic RAG pipelines
Quick prototypes
No cycles needed

Use LangGraph

Loops & retries
Multi-agent systems
Human-in-the-loop
Persistent state

KEY DECISION POINTS

🔄 Need cycles/loops? → LangGraph

💾 Need checkpointing? → LangGraph

👥 Multiple agents? → LangGraph

⚡ Simple chain? → LangChain LCEL

Q20 / 20

HARD

Best practices for production LangGraph apps?

💡 **ANSWER**

Production LangGraph applications require careful attention to state design, error handling, persistence, and observability.

PRODUCTION CHECKLIST

- ✅ **Persistent Checkpointer:** Use Postgres/MongoDB, not InMemorySaver
- ✅ **Error Handling:** Wrap nodes in try/catch, use fallback edges
- ✅ **Timeouts:** Set max iterations to prevent infinite loops
- ✅ **Observability:** Use LangSmith for tracing & debugging
- ✅ **State Validation:** Validate state at node boundaries

⚠️ **COMMON PITFALL**

Always set recursion_limit to prevent runaway agent loops!

Thank You for Reading! 🎉

Naved Khan

Gen AI Engineer

💾 *Save this for interview prep.*

🔄 *Share with someone who needs it!*

❤️ Like 💬 Comment 🔄 Repost 🔖 Save