PLURALIT
AI SOLUTIONS

MODULE 3 UNIT 1.1

LangGraph theory and basic concepts

Ver. 1.0

Table of contents

1. LangGraph theory and basic concepts

1.1 Learning goals

By the end of this unit you can:

- Explain why a **graph** (explicit control flow) beats ad-hoc loops for complex agents.
- Describe **nodes**, **edges**, **state**, **messages**, **tools**, and **checkpointing**.
- Choose appropriate **reducers** per state field and justify the choice.
- Sketch and reason about exit conditions for small agentic DAGs (router, tool loop, finalize).
- Identify at least three **failure cases** (tool loop runaway, empty retrieval, schema mismatch) and propose guards.

1.2 What LangGraph is — and isn't

Is: A framework for building directed acyclic graphs (DAGs) of steps with a typed, shared state. Nodes may call LLMs, execute pure code, or invoke tools. Edges (static or conditional) determine what runs next. Built-in checkpointing makes runs replayable and resumable.

Isn't: A black-box agent that decides everything at runtime. You, the designer, constrain where branching may occur, which tools are allowed, and how/when to exit.

Use when… you need auditability, guardrails, and explicit business logic (validation, loop caps, approvals).

Avoid when… a simple, linear chain suffices.

1.3 Core concepts (deep dive)

1.3.1 Graphs, nodes & edges

- **Graph (DAG):** START → … → END. Apparent "loops" are modeled via **conditional edges** that jump back only while a guard is true.

- **Node:** Pure function-like step node(state) → partial_update. May call an LLM, run Python, or orchestrate subgraphs.

- **Edge:** Transition rule. **Static edges** always go to a specific node; **conditional edges** inspect state (or model output) to choose the next node.

Design tip: Keep nodes **small and single-purpose**. Test them as pure functions that read state and return a delta.

1.3.2 State & reducers (designing your schema)

State is a typed mapping every node **reads** and **writes**. Each field declares a **reducer** that defines how multiple updates merge across steps. Pick reducers that match intent:

Field	Type	Reducer	Why
messages	list[BaseMessage]	add_messages (append)	preserves full dialog + tool IO
plan	str	overwrite	keep the latest plan only
tool_calls	int	operator.add	simple loop cap & telemetry
context_docs	list[Doc]	append or set	accumulate evidence vs. replace
errors	list[str]	list concat	post-mortem friendly
route	Literal[...]	overwrite	one active route at a time
scratch	dict	shallow merge	ephemeral per-step scratchpad

Reducer principles

- **Append** for histories/logs.
- **Overwrite** for singletons (current plan/route).
- **Numeric add** for counters (tool calls, retries).
- Avoid storing large blobs; store **references** (ids/keys) and fetch on demand.

1.3.3 Messages & tool-calling semantics

Messages carry dialog between **Human**, **AI**, and **Tools**. When the model wants a tool, it emits a **tool call** (name + args). After execution, you append a **ToolMessage** referencing the tool-call id. The add_messages reducer keeps the chain of **request → result** intact so the next LLM step can reason over it.

Gotchas

- Returning raw strings instead of ToolMessage breaks the call/response chain.
- Dropping tool_call_id prevents the model from associating outputs to its request.
- Binding too many tools at once increases confusion; prefer **local allow-lists** per node.

1.3.4 Tool nodes & routing with `tools_condition`

- Bind tools at the node: llm.bind_tools([my_tool, ...]).
- After an LLM node, add_conditional_edges(node, tools_condition) routes to **ToolNode only** when a tool was requested; otherwise continue/exit.
- After tools run, edges usually return to the LLM node to let it incorporate results, or route onward to a finisher.

1.3.5 Checkpointing, threads & resumability

- **Checkpointing** persists state deltas between nodes — enabling **replay**, **resume**, and audit trails.
- Use a **thread id** (e.g., session id) so a conversation/run can be resumed later.

- For development, MemorySaver suffices; for durability, use SqliteSaver or a custom backend.

1.3.6 Human-in-the-loop & interrupts (overview)

Place **interrupts** at critical edges (e.g., before a payment or external mutation). Execution pauses with a checkpoint; a human reviews/edits state; the run resumes deterministically.

1.4 Execution lifecycle (step-by-step)

1. **Initialize state** (seed messages, counters, scratch).
2. Enter at **START** → first node (often agent).
3. **Agent node** runs; it may emit a tool call.
4. **Route**: if tool requested → **ToolNode**; else continue/exit.
5. **ToolNode** executes → append **ToolMessage** with results.
6. **Loop or exit**: go back to agent for further reasoning, or follow static edges toward END.
7. **Checkpoint** between nodes for replay/resume.

> Debugging tip: stream events during execution to observe node transitions and state deltas in real time.

1.5 Canonical patterns & when to use them

- **Single-LLM + Tools (loop with cap)** — Small assistants with a handful of deterministic tools. Maintain tool_calls counter to prevent runaway.
- **Router → Specialists** — Natural task modes (retrieval, math, summarize). Keep tools local to each specialist to reduce ambiguity.
- **Analyze → Act → Reflect → Finalize** — Separate planning from acting; add a reflective check before final output.
- **Guarded Retry** — For flaky APIs: increment retries; branch while retries < N; otherwise give_up with a clear error.

1.6 Design checklist (production-minded)

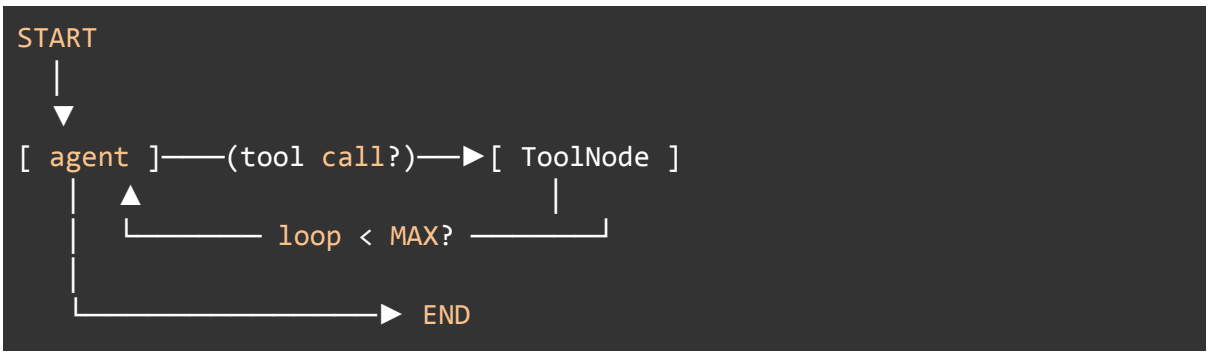
- **Exit conditions** are explicit and testable (no unbounded loops).
- **Reducers** reflect semantics (append vs overwrite vs add); avoid mixing meanings.
- **Local tool allow-lists**; validate inputs before execution.
- **Observability** on: checkpointing; traces; sensitive fields redacted.
- **Error policy**: capped retries; user-visible errors are clear; errors list maintained.
- **Performance**: keep state small; store ids instead of large payloads; fetch as needed.
- **Determinism for tests**: temperature 0; seed when supported; unit-test nodes as pure functions.

Anti-patterns

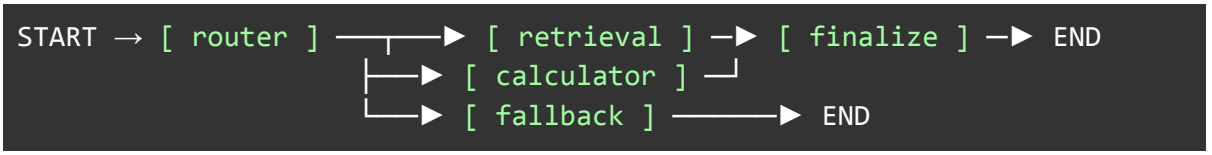
- A single “god” node that plans, routes, calls tools, and finalizes.
- Binding every tool to every node.
- Treating state as a junk drawer (bloated checkpoints; noisy diffs).

1.7 Concept diagrams (ASCII)

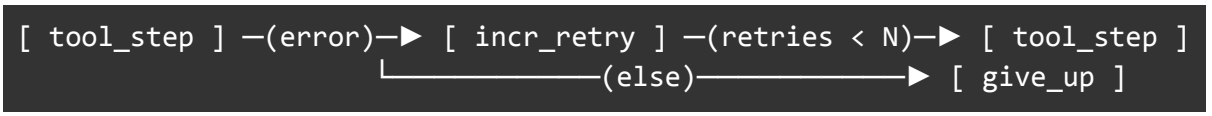
a) Single-LLM + Tools (loop with cap)



b) Router to specialists



c) Guarded retry



1.8 Cookbook: schema & edge-condition templates

State schema snippets

```
messages:   Annotated[List[BaseMessage], add_messages]   # full IO
history
plan:        str                                          # overwrite
route:       Literal["retrieval","math","finalize"]      # overwrite
tool_calls: Annotated[int, operator.add]                  # count
retries:    Annotated[int, operator.add]                  # errors
errors:     Annotated[List[str], list.__add__]            # concat
```

```
context_ids: List[str]                                    # ids, not
blobs
scratch:     Dict[str, Any]                               # ephemeral
```

Edge conditions (pseudocode)

```
# After agent node
if model_requested_tool(state.messages):
    goto ToolNode
elif state.route == "finalize":
    goto finalize
else:
    goto router

# Guarded loop cap
if state.tool_calls >= MAX_TOOL_CALLS:
    goto safe_exit
else:
    goto agent

# Retry policy
if state.retries < 3 and last_error_is_transient(state):
    goto retry_node
else:
    goto give_up
```

1.9 Observability, safety & performance

- **Checkpointing** + consistent thread_id for replay/resume across user sessions.
- **Tracing**: capture inputs/outputs of nodes and tools; redact secrets early.
- **Safety**: restrict tools per node; validate arguments; interpose human approval where needed.
- **Performance**: minimize state size; avoid embedding large docs in state; prefer ids/handles; cache deterministic tool results.

1.10 FAQ & common pitfalls

- **"My graph loops forever."** Add a counter (tool_calls) and a loop-cap edge. Ensure router edges are mutually exclusive and total.
- **"Tool output isn't visible to the model."** Make sure you append a ToolMessage tied to the original tool_call_id.
- **"State keeps growing."** Use overwrite reducers for singletons and store ids instead of large payloads.
- **"Hard to debug what happened."** Enable checkpointing and stream events to view per-node transitions and deltas.

1.11 Glossary

- **Reducer**: merge strategy for a state field (append, overwrite, numeric add).
- **ToolNode**: executes tool functions requested by the LLM (via function-calling).
- **Checkpoint**: persisted snapshot of state between node executions.
- **Thread id**: grouping key (e.g., session id) for runs; enables resume/replay.
- **Interrupt**: deliberate pause requiring human action before proceeding.
- **Conditional edge**: runtime decision that picks the next node based on state/model output.

1.12 Deepen Your Understanding (Required)

Before moving forward, let's deepen our collective understanding. This section contains the required material designed to ensure you have a robust grasp of the core concepts. A full understanding here is mandatory to successfully tackle the complexities ahead.

- Basic LangGraph for beginners
- Beginners Tutorial

Appendices

Appendix 1: Backup links to external resources

- Basic LangGraph for beginners
- Beginners Tutorial