AIBees Academy

Agentic AI

Complete Student Notes

LangChain | LangGraph | Agents | RAG | MCP | Observability

Learning Roadmap

Work through these chapters in order. Each builds directly on the previous one.

Chapter	Topic	What You Will Be Able To Do
1	From LangChain to LangGraph	Understand why stateful agents need a graph, not a chain
2	Agentic Frameworks Compared	Choose the right framework for any project
3	Workflow to Agent to Multi-Agent	Build progressively complex agentic systems
4	Agentic RAG vs Traditional RAG	Implement intelligent iterative retrieval
5	LangGraph Deep Dive	Use graphs, routers, ReAct, orchestrators, reducers, sub-graphs
6	MCP - Model Context Protocol	Connect agents to external tools and services
7	Observability and Langfuse	Monitor, debug, and optimise every agent run

Chapter 1

From LangChain to LangGraph

Why chains are not enough for real agents

1.1 What is LangChain?

LangChain was the first widely-adopted framework for building applications on top of Large Language Models. Released in 2022, it introduced a powerful idea: chain LLM calls together with tools, memory, and data sources using a common interface. Almost overnight, it became the standard starting point for LLM-powered applications.

Think of LangChain as a toolkit. It provides ready-made building blocks:

- **LLM wrappers** - call OpenAI, Gemini, Anthropic, or any local model with the same Python interface
- **Prompt templates** - reusable, parameterised prompts with input variables
- **Chains** - pipe the output of one LLM call directly into the next step
- **Tools** - Python functions the LLM can call (web search, calculator, database queries)
- **Memory** - store conversation history so the LLM can refer back to earlier turns
- **Agents (basic)** - let the LLM choose which tool to call next using ReAct prompting

Analogy: LangChain as a Factory Assembly Line

LangChain is like a factory assembly line. Raw material (your prompt) goes in one end, passes through each station (LLM call, tool, LLM call) in a fixed sequence, and the finished product (your answer) comes out the other end.

It works perfectly when the steps are known in advance and the path never changes.

1.2 LangChain Limitations

LangChain is excellent for linear pipelines, but real-world agentic applications are rarely linear. The limitations became clear as developers tried to build more complex systems:

Limitation	What Goes Wrong	Example
No state management	State lives inside each chain call - not persisted between steps	Agent forgets what it did 3 steps ago
No branching or loops	Chains flow one direction - no conditional routing, no retries	Cannot re-search if first search found nothing

Limitation	What Goes Wrong	Example
No human-in-the-loop	Cannot pause mid-execution to ask a human for input	Cannot checkpoint and resume a 10-step analysis
No parallel execution	Steps run sequentially even when independent	Three database queries run one by one
Agent instability	LCEL (Langchain Expression Language) agents can loop unpredictably without guards	ReAct agent enters an infinite tool-call loop
Hard to debug	No built-in graph visualisation or step tracing	Impossible to know which step produced wrong answer
Tight coupling	Changing one step often breaks surrounding steps	Adding a new tool breaks the chain structure

The Core Problem with Chains

LangChain's LCEL chains are fundamentally stateless pipelines.
When you need an agent that remembers what it did, decides what to do next based on results, and can loop until it succeeds - a chain becomes the wrong tool for the job.
This is exactly the problem LangGraph was designed to solve.

1.3 What is LangGraph?

LangGraph is a framework built on top of LangChain that models agent behaviour as a directed graph. Instead of a linear pipeline, your application is a set of nodes (functions) connected by edges (routing rules). The agent's current status is stored in a shared state object that every node can read and update.

LangGraph was created specifically to solve the limitations above. It gives you:

- **Persistent shared state** - a TypedDict that lives across all node executions
- **Conditional edges** - route to different nodes based on the current state
- **Cycles and loops** - a node can route back to an earlier node safely
- **Human-in-the-loop** - pause execution at any node and wait for human approval
- **Parallel execution** - run multiple nodes simultaneously
- **Sub-graphs** - nest entire graphs inside a parent graph for modular design
- **Streaming** - stream intermediate results node by node to the UI
- **Checkpointing** - save and restore state so long-running agents can be resumed

1.4 LangChain vs LangGraph - Side by Side

Dimension	LangChain (LCEL)	LangGraph
Mental model	Assembly line - linear steps	State machine - graph of decisions
State	Stateless between steps	Persistent TypedDict across all nodes
Routing	Fixed - always A then B then C	Dynamic - conditional edges based on state
Loops	Not supported	First-class with safety guards
Human-in-loop	Not supported	Built-in interrupts and checkpoints
Parallel	Not supported	Supported via Send API
Debugging	Print statements	Graph visualisation plus Langfuse tracing
Best for	Q&A chatbots, simple RAG	Agents, multi-agent, production AI

Which One to Use?

Use LangChain LCEL for: simple chatbots, one-shot Q&A, linear RAG pipelines.
Use LangGraph for: agents that loop, multi-agent systems, human-in-the-loop workflows, anything that needs to remember what it did and decide what to do next.

The good news: LangGraph is built ON TOP of LangChain - you use both together.
LangChain provides the LLM wrappers, prompt templates, and tools.
LangGraph provides the graph, state management, and control flow.

Chapter 2

Agentic Frameworks Compared

CrewAI | LangGraph | AutoGen | Semantic Kernel | n8n

2.1 The Landscape

Several frameworks now compete to help developers build agentic AI systems. Understanding what each one optimises for will help you choose the right tool for any project - and understand what LangGraph uniquely provides that others do not.

CrewAI

CrewAI models agents as a team (a crew) of specialists. You define agents with specific roles, goals, and backstories, then assign them tasks. CrewAI handles the orchestration automatically. It is the fastest way to get a multi-agent system running.

- **Strength:** Extremely easy to build role-based multi-agent systems in minutes
- **Strength:** Great for content generation, research pipelines, document workflows
- **Limitation:** You can only use agents in the way CrewAI defines them - role, goal, backstory, task
- **Limitation:** Cannot build custom agent architectures or custom graph structures
- **Limitation:** Limited control over execution flow - CrewAI decides how agents communicate
- **Limitation:** Harder to debug - orchestration is largely hidden from the developer

CrewAI: What You Control vs What You Cannot

What CrewAI gives you: Role-based agents, task assignment, automatic handoffs
What you control: Agent roles, task descriptions, tool list per agent
What you cannot change: How agents communicate, memory sharing, execution order

Analogy: CrewAI is a pre-built office where you can hire people for specific roles.
LangGraph lets you design the office, org chart, and communication channels.

LangGraph

LangGraph models agents as a graph of nodes and edges. You define exactly what each node does, how state flows between nodes, and what conditions trigger different routes. It is the most flexible and powerful option available today.

- **Strength:** Full control - you build any agent architecture you can imagine

- **Strength:** Custom agents from scratch - not limited to pre-defined patterns
- **Strength:** First-class support for loops, human-in-the-loop, parallel execution
- **Strength:** Production-grade - used by Anthropic, Google, and major enterprises
- **Limitation:** Steeper learning curve - you must understand graphs, state, and edges
- **Limitation:** More boilerplate for simple use cases where CrewAI would be faster

AutoGen (Microsoft)

AutoGen focuses on multi-agent conversations. Agents are modelled as participants in a chat - they send messages to each other and decide who speaks next. It is natural for debate-style or review-style workflows.

- **Strength:** Natural for debate, review, or critique multi-agent workflows
- **Strength:** Strong integration with code execution agents
- **Limitation:** Conversation metaphor becomes awkward for non-conversational workflows
- **Limitation:** Less control over state management than LangGraph

Semantic Kernel (Microsoft)

Semantic Kernel is a production-focused SDK for integrating LLMs into enterprise applications. It supports C#, Python, and Java and focuses on plugin-based architecture for Microsoft ecosystem integration.

- **Strength:** Enterprise integrations - Azure, Office 365, Microsoft Graph
- **Strength:** Multi-language support (Python, C#, Java)
- **Limitation:** Less flexible for custom agentic architectures
- **Limitation:** Heavier than LangGraph for pure Python agentic use cases

n8n

n8n is a low-code / no-code workflow automation platform. It allows non-developers to connect AI models with external services using a visual drag-and-drop interface. It is not a developer framework.

- **Strength:** No code required - visual workflow builder
- **Strength:** Hundreds of integrations out of the box
- **Limitation:** Not suitable for custom agentic logic - limited to what the UI supports
- **Limitation:** No programmatic control over agent behaviour

2.2 Framework Decision Guide

Use Case	Best Framework	Why
Quick multi-agent content pipeline	CrewAI	Role-based setup is fast, no graph knowledge needed
Full control over agent architecture	LangGraph	Custom graphs, custom state, custom routing
Enterprise integrations (Azure, Office)	Semantic Kernel	Built for Microsoft enterprise ecosystem
Multi-agent conversation or review	AutoGen	Conversation-first architecture fits naturally
No-code automation with AI	n8n	Visual builder, no programming required
Production agentic AI at scale	LangGraph	State management, checkpointing, observability
RAG plus Agent plus Human-in-the-loop	LangGraph	All three are first-class features

Chapter 3

Workflow | Agent | Multi-Agent

The three levels of agentic complexity

3.1 The Spectrum of Agentic AI

Agentic AI is not a single thing - it is a spectrum. As you move along the spectrum, the system gains more autonomy, more capability, and more complexity. Understanding where each level sits helps you choose the right architecture for any problem.

The Agentic Spectrum

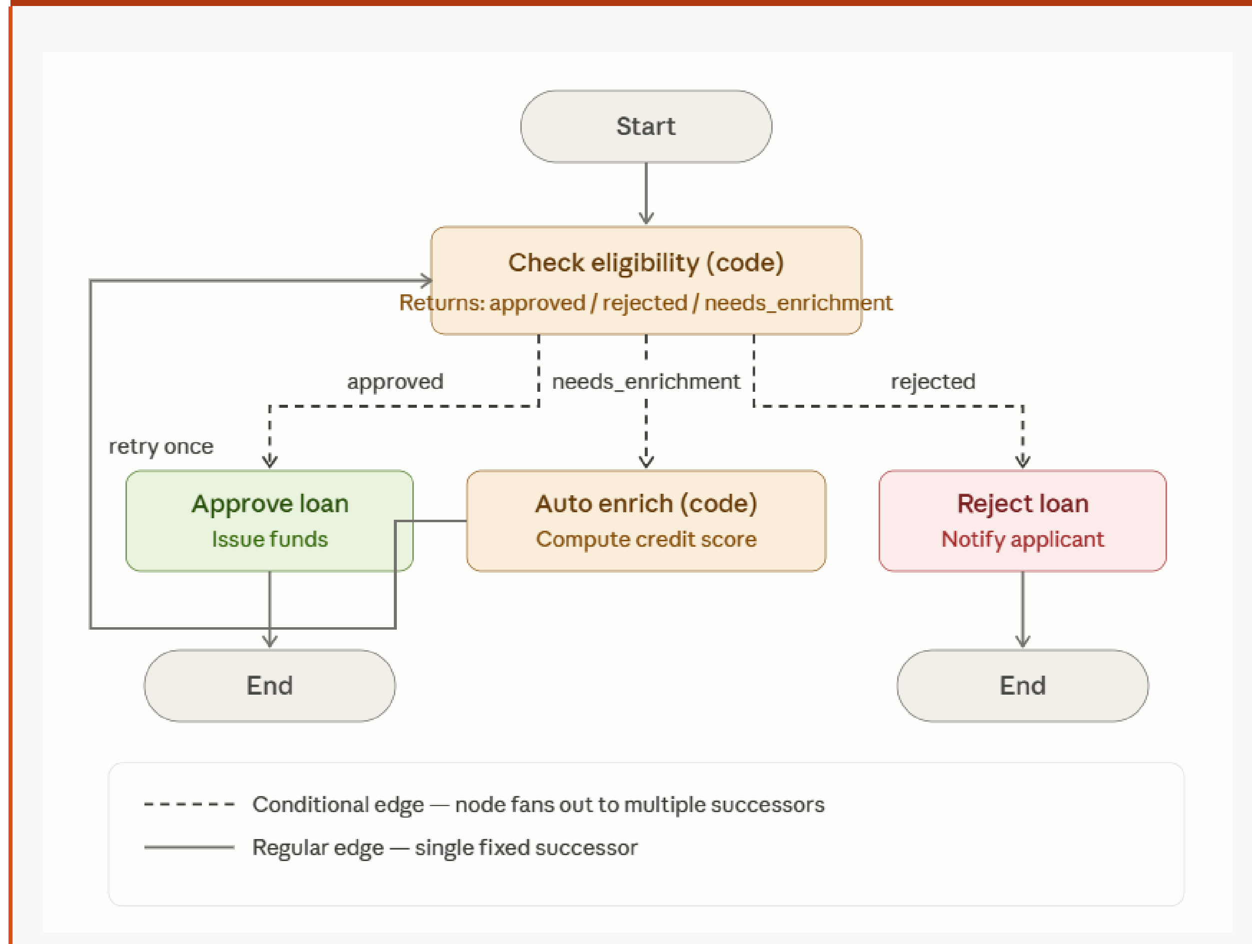
```
LEAST AUTONOMOUS <--------------------------------> MOST AUTONOMOUS

   Workflow              AI Agent              Multi-Agent (Agentic AI)
   ----------------     ------------------     ----------------------------
   Fixed steps          LLM decides steps      Multiple LLMs coordinate
   No LLM routing       One LLM + tools        Each agent is a specialist
   Predictable          Adaptive               Emergent intelligence
   Easy to debug        Harder to predict      Requires orchestration

   Example:             Example:               Example:
   Loan screening        Fraud analyst           Hospital bed allocator
   (fixed rules)        (picks tools            (5 specialist agents +
                         per transaction)        supervisor)
```

3.2 Level 1 - Workflow

A workflow is a pre-defined sequence of steps. The order is fixed at design time. An LLM may be used at specific steps for decisions or generation, but it does not control the flow - the developer does.

- **The developer decides:** which steps run, in what order, and which step comes next
- **The LLM's role:** evaluate data and return a decision (approve / reject / needs more info)
- **Key characteristic:** if you draw the graph, you see a fixed path from START to END
- **When to use:** when the process is well-understood, rules-based, and predictable

Workflow Graph - Loan Screening Example



In this workflow, the LLM evaluates the loan application. If it needs a credit score, the system computes it and runs the LLM again. But the path - check, enrich, check, approve or reject - is fixed by the developer, not decided by the LLM at runtime.

Workflow Characteristics

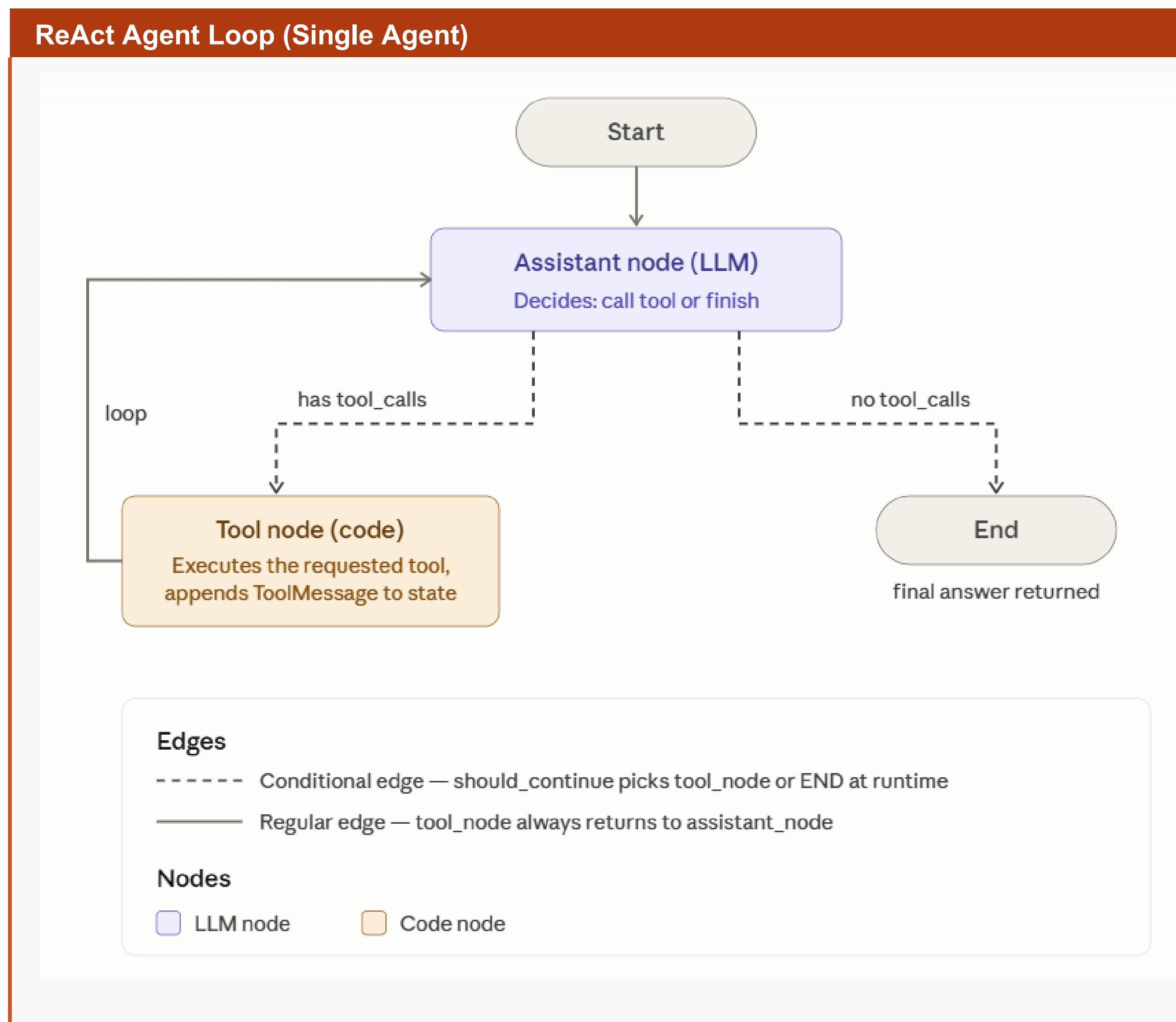
YES Pre-defined node order - no runtime routing decisions by the LLM
YES Deterministic - same input always follows the same path
YES Easy to test and debug
YES LLM is used for evaluation, not for routing
NO Cannot adapt to unexpected situations
NO Cannot call tools it was not told to call

3.3 Level 2 - AI Agent (Single Agent)

An AI agent uses an LLM to decide at runtime which action to take next. The LLM reads the current state (messages, tool results so far) and decides: call a tool, call a different tool, or produce the final answer. The agent loop continues until the LLM stops calling tools.

- **The LLM decides:** which tool to call, in what order, and when to stop
- **Tools available:** a set of @tool-decorated functions the LLM can invoke by name
- **State:** a messages list - every tool call and result is appended to the history
- **Key characteristic:** the execution path is NOT fixed - it is different for every input
- **When to use:** when different inputs need different investigation paths

ReAct Agent Loop (Single Agent)



Example - Fraud Analyst Agent: For a high-velocity transaction in a foreign city, the agent calls check_transaction_velocity, then check_geo_anomaly, then calculate_risk_score, then

classify_transaction. For a routine low-value transaction, it may call only 2 tools and stop. The path is decided at runtime by the LLM based on what the data shows.

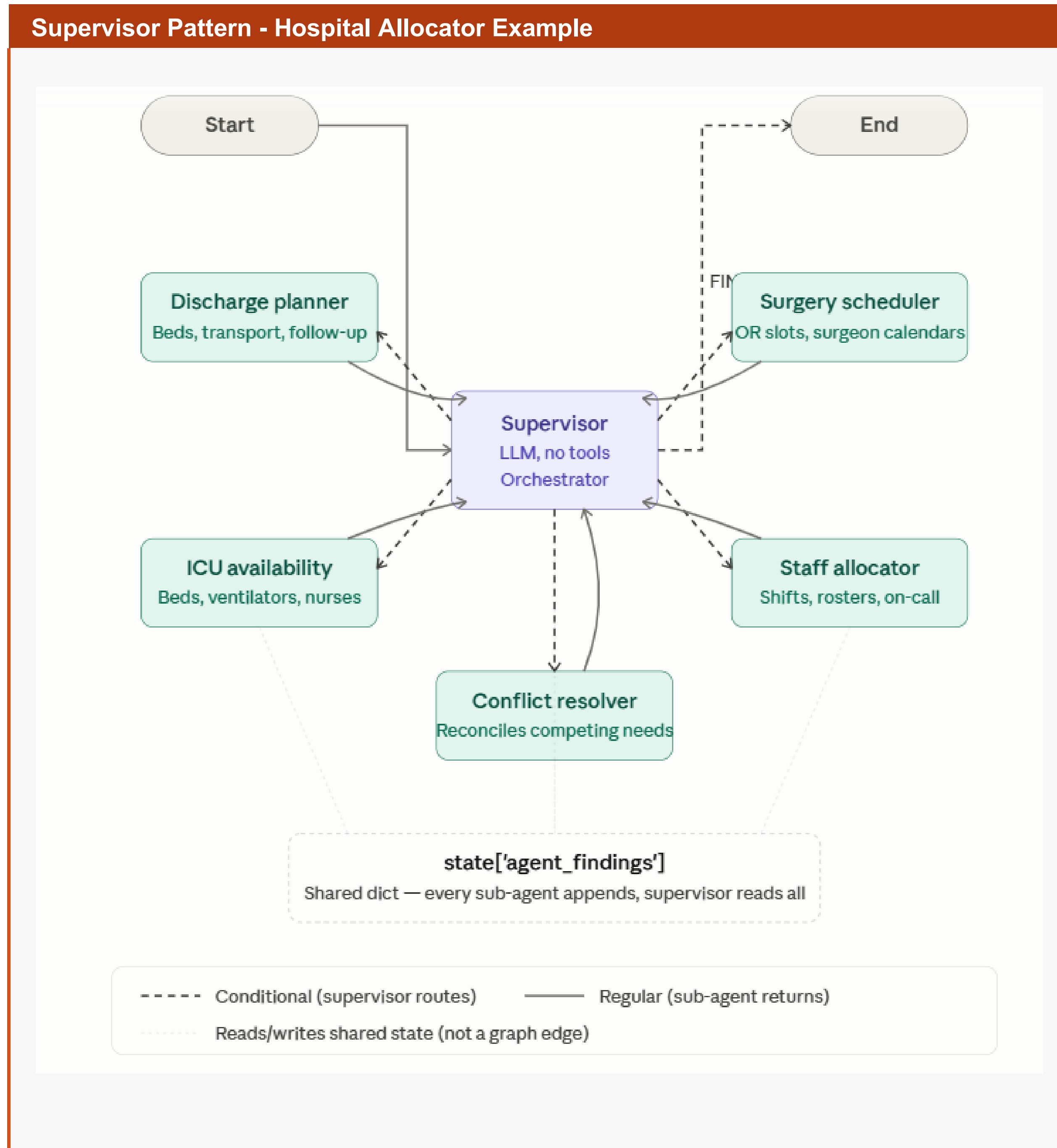
	Workflow	AI Agent
Who decides next step?	Developer (hardcoded)	LLM (at runtime)
Execution path	Same for every input	Different for every input
Tools used	Same every run	LLM picks which tools and how many
State structure	Named TypedDict fields	messages list with add_messages reducer
Routing mechanism	Conditional on field values	should_continue checks for tool_calls
Loop mechanism	Back-edge with counter guard	Agent loop until LLM stops calling tools

3.4 Level 3 - Multi-Agent (Agentic AI)

Agentic AI refers to systems where multiple agents coordinate to solve a problem that no single agent could handle efficiently on its own. Each agent is a specialist with its own LLM, its own tools, and its own system prompt. An orchestrator (supervisor) coordinates the specialists.

- **Why multiple agents?** Different domains need different tools. Giving one agent all the tools causes confusion - the LLM may pick the wrong tool for a domain it was not designed for.
- **Supervisor pattern:** A supervisor agent reads the accumulated findings and decides which specialist to call next. It does NOT have tools - it only orchestrates.
- **Star topology:** All sub-agents route back to the supervisor after completing their work. Only the supervisor can decide to END.
- **agent_findings:** Each sub-agent writes its findings into a shared dict so the supervisor can read all results before making the final plan.

Supervisor Pattern - Hospital Allocator Example



	Single Agent	Multi-Agent (Supervisor)
Number of LLMs	1 LLM with all tools	1 supervisor + N specialist LLMs
Tools	All tools in one list	Each agent has domain-specific tools only
Routing	should_continue (tool_calls?)	route_supervisor (next_agent string)

	Single Agent	Multi-Agent (Supervisor)
State	messages + iteration_count	messages + next_agent + agent_findings
Supervisor	Not applicable	No tools - only reads and orchestrates
Graph topology	Linear loop	Star - all agents route back to supervisor

When to Go Multi-Agent

Go multi-agent when:
- The problem spans multiple distinct domains (ICU, surgery, staffing)
- No single agent can hold all the tools without getting confused
- Different parts of the problem require different expertise
- You need parallel investigation followed by consolidated decision-making

Stay single-agent when:
- The problem domain is coherent (all about fraud, all about one patient)
- 6 to 8 tools is manageable for one LLM
- Sequential tool use is sufficient

Chapter 4

Agentic RAG vs Traditional RAG

From single retrieval to iterative reasoning

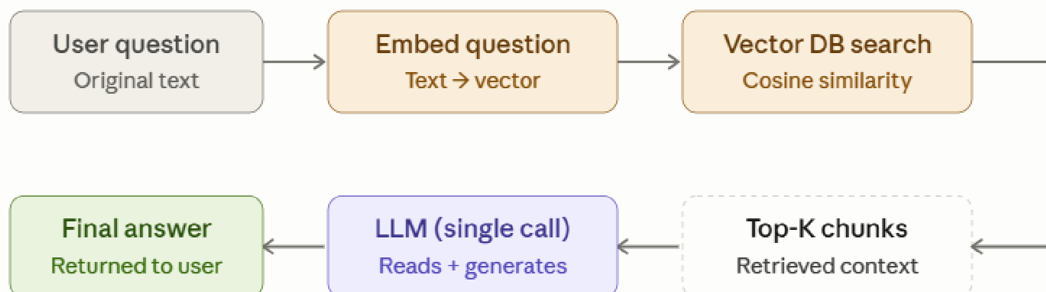
4.1 Traditional RAG - Recap

RAG (Retrieval-Augmented Generation) was introduced to solve the knowledge cutoff problem: LLMs know only what they were trained on. RAG connects an LLM to a fresh, updatable knowledge base so it can answer questions about current documents, private data, and real-time information.

The traditional RAG pipeline:

- **Embed the user's question into a dense vector**
- **Search the vector database for the most similar chunks**
- **Inject those chunks into the LLM prompt as context**
- **LLM reads the context and generates an answer**

Traditional RAG - One Pass



Hardcoded behavior

Number of searches: always exactly 1

Search query: always the original user question (no rewriting, no decomposition)

4.2 Why Traditional RAG Fails on Complex Questions

Consider this question: 'Is it safe to prescribe Ibuprofen to Suresh Babu for his gout pain?'

Traditional RAG searches 'Ibuprofen gout pain', retrieves gout treatment guidelines, and the LLM says: 'Yes, NSAIDs are effective for acute gout.'

WRONG - and potentially dangerous

Suresh Babu has CKD Stage 4 (eGFR 24 mL/min).
NSAIDs in CKD Stage 4 cause acute kidney failure - they are ABSOLUTELY CONTRAINDICATED.

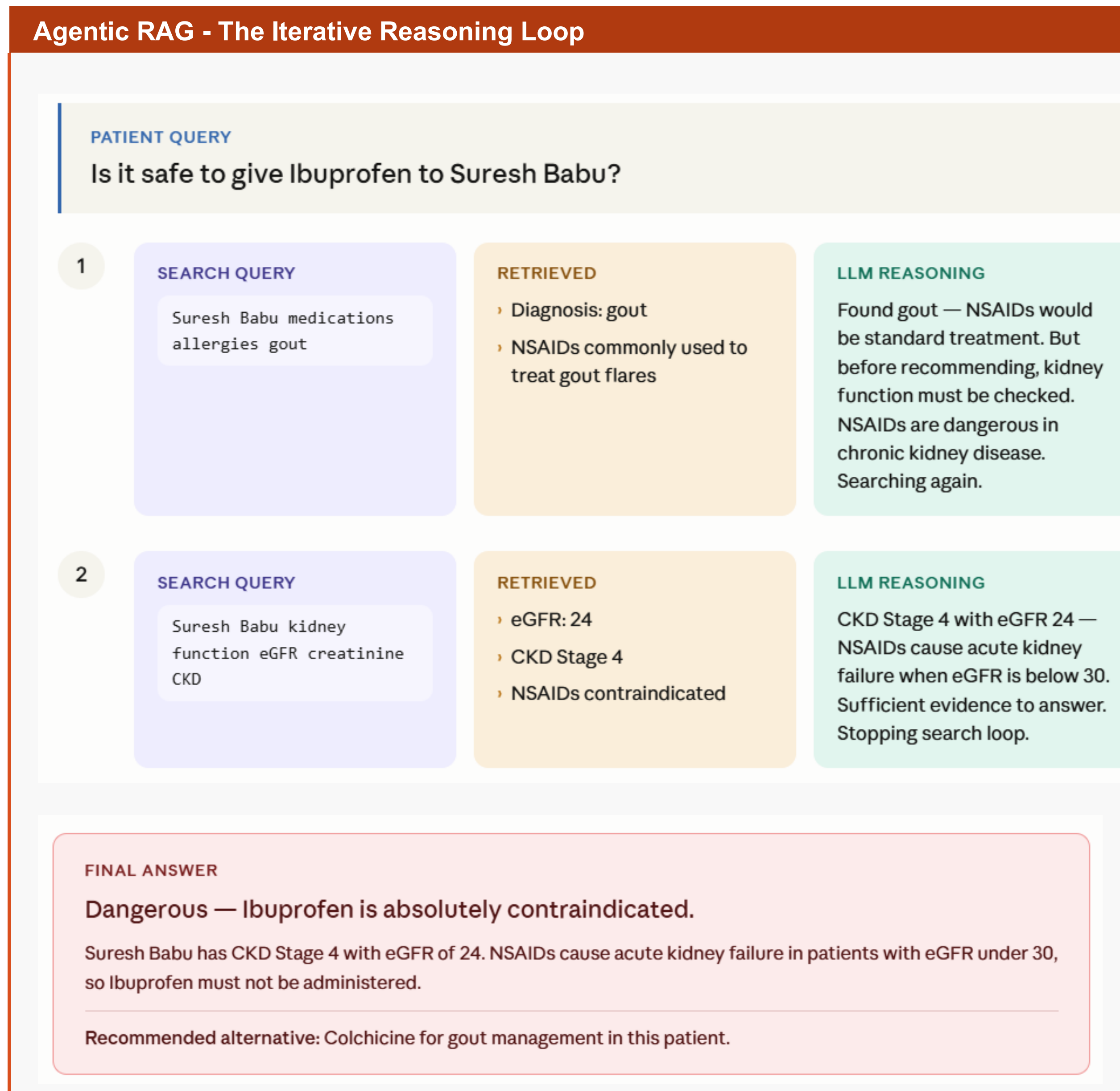
Traditional RAG searched once. It found gout treatment information. It never checked kidney function. The answer is medically incorrect and could harm the patient.

The problem: traditional RAG always searches exactly once using the original question. It cannot recognise that the first search found a condition requiring a safety check, and go looking for that safety information automatically.

4.3 Agentic RAG - Iterative Retrieval with Reasoning

Agentic RAG replaces the single search-and-answer with an intelligent loop. The LLM reads the retrieval results and reasons: 'Do I have enough to answer safely? Or do I need to search for something else first?' It continues searching until it is confident it has complete, safe information.

Agentic RAG - The Iterative Reasoning Loop



The second search only happened because the first search revealed a condition that triggered a follow-up. This is the core of Agentic RAG - the LLM is in the reasoning loop, not just the answering loop.

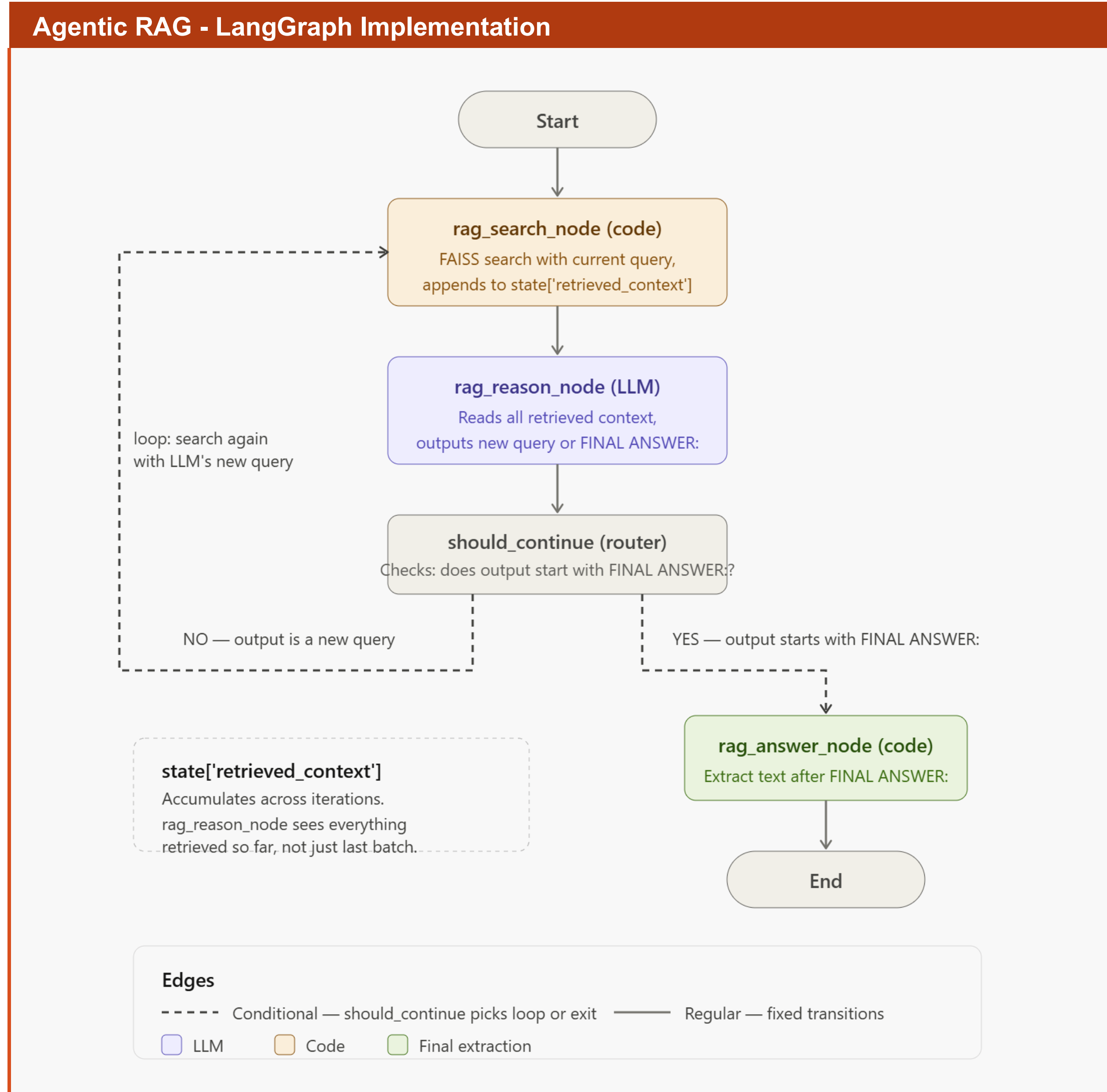
Dimension	Traditional RAG	Agentic RAG
Number of searches	Always exactly 1	1 to N - LLM decides when to stop
Search query source	Always the original question	LLM writes custom queries each round
Second search trigger	Never	LLM decides based on what first search found

Dimension	Traditional RAG	Agentic RAG
Cross-document reasoning	Limited - one search, one set of chunks	Strong - builds context across searches
Complex medical questions	May miss contraindications	Systematically checks all relevant factors
Implementation	Simple: embed, search, answer	LangGraph loop: search, reason, search, answer
Token cost	Low	Higher (multiple LLM calls)
Accuracy for complex Q	Moderate	Significantly higher

4.4 Agentic RAG in LangGraph

The key difference in implementation: Agentic RAG is a proper LangGraph StateGraph with a conditional loop. Traditional RAG is just a function call. Because it is a compiled StateGraph, Langfuse shows the entire reasoning loop as a graph visualisation.

Agentic RAG - LangGraph Implementation



Simple RAG vs Agentic RAG - Graph Difference

```
Simple RAG graph:   START -> rag_search -> rag_simple -> END
                (linear, no loop, one search, one LLM call)

Agentic RAG graph:  START -> rag_search -> rag_reason -> should_continue
                                    |
                         search again? -> rag_search
                         final answer? -> rag_answer -> END

Langfuse trace for Simple RAG:   __start__ -> rag_search -> rag_simple -> __end__
Langfuse trace for Agentic RAG:  __start__ -> rag_search -> rag_reason ->
                           rag_search -> rag_reason -> rag_answer -> __end__
                           (the loop is visually obvious in the graph)
```

Chapter 5

LangGraph Deep Dive

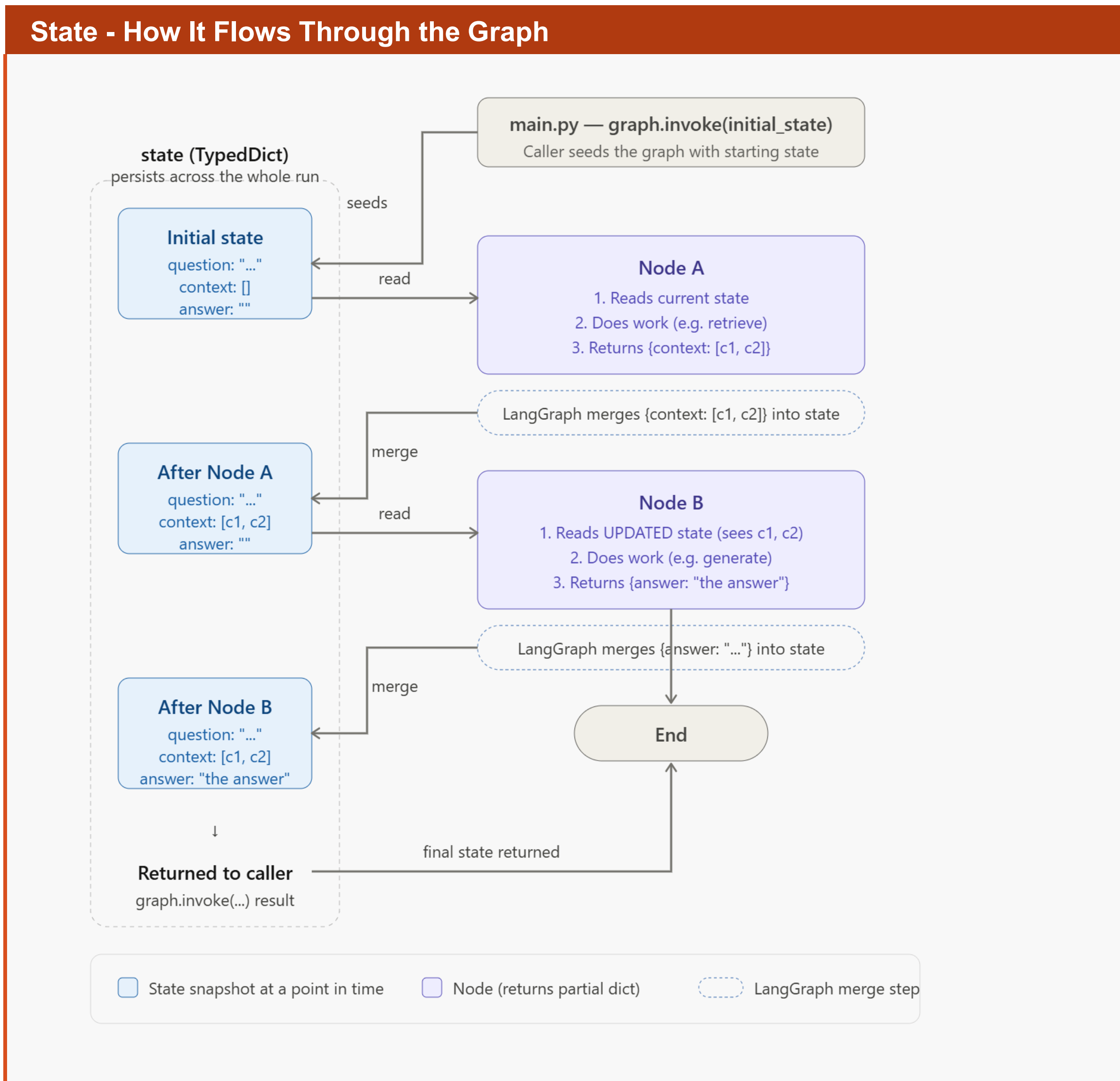
Graphs | Routers | ReAct | Orchestrator | Reducer | Sub-graphs

5.1 Core Building Blocks

State - The Heart of Every LangGraph Application

Every LangGraph application begins with a state definition. State is a TypedDict - a Python dictionary with type annotations. Every node reads from state and writes updates back to state. State persists across the entire graph execution.

State - How It Flows Through the Graph



State fields use reducers. The default reducer overwrites a field when a node returns an update. The special add_messages reducer **APPENDS** to a list rather than replacing it. This is what gives agents memory across tool calls.

State Definition - Fraud Analyst Agent

```
class FraudState(TypedDict):
    messages:       Annotated[list[BaseMessage], add_messages]  # APPEND
    current_txn_id: Optional[str]                               # OVERWRITE
    verdicts:       dict                                        # OVERWRITE
    iteration_count: int                                        # OVERWRITE

  messages uses add_messages -> every node appends new messages to the list
  All other fields use the default reducer -> last write wins
```

Nodes

A node is any Python function that takes state as input and returns a dict of updates. Nodes do the actual work: call LLMs, execute tools, fetch data, and make decisions.

- **Rule:** nodes must return a dict - not the full state, just the fields they changed
- **Rule:** nodes should do one thing - single responsibility makes them testable
- **Rule:** nodes should not call each other - the graph handles routing

Edges

Edges define how the graph flows between nodes. There are three types:

- **Static edge:** always go from A to B - builder.add_edge(A, B)
- **Conditional edge:** go to A or B based on state - builder.add_conditional_edges(source, router_fn)
- **Entry and exit edges:** add_edge(START, first_node) and add_edge(last_node, END)

5.2 Graph Patterns

Pattern 1 - Simple Linear Graph

The simplest graph: a sequence of nodes with no branching. Every input follows the same path. Use this for deterministic pipelines where every step must always run.

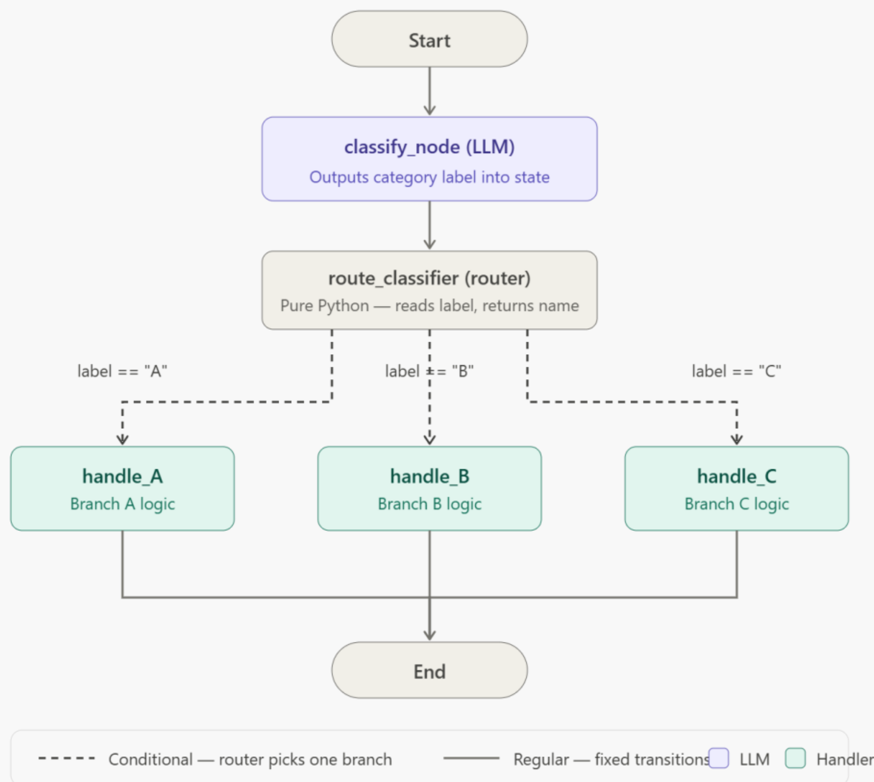
Simple Linear Graph

```
START --> node_a --> node_b --> node_c --> END

builder.add_edge(START,    'node_a')
builder.add_edge('node_a', 'node_b')
builder.add_edge('node_b', 'node_c')
builder.add_edge('node_c', END)
```

Pattern 2 - Router

A router is a conditional edge that reads state and decides which of several nodes to route to next. The router function is pure Python - no LLM needed. It reads state and returns a string (the name of the next node).

Router Pattern - Conditional Branching




```
def route_classifier(state) -> str:
    return state['category']    # 'A', 'B', or 'C'

builder.add_conditional_edges('classify_node', route_classifier,
    {'A': 'handle_A', 'B': 'handle_B', 'C': 'handle_C'})
```

Router Function vs Conditional Edge

Router function: pure Python you write - reads state, returns a string (next node name).
Conditional edge: LangGraph mechanism that calls your router function.

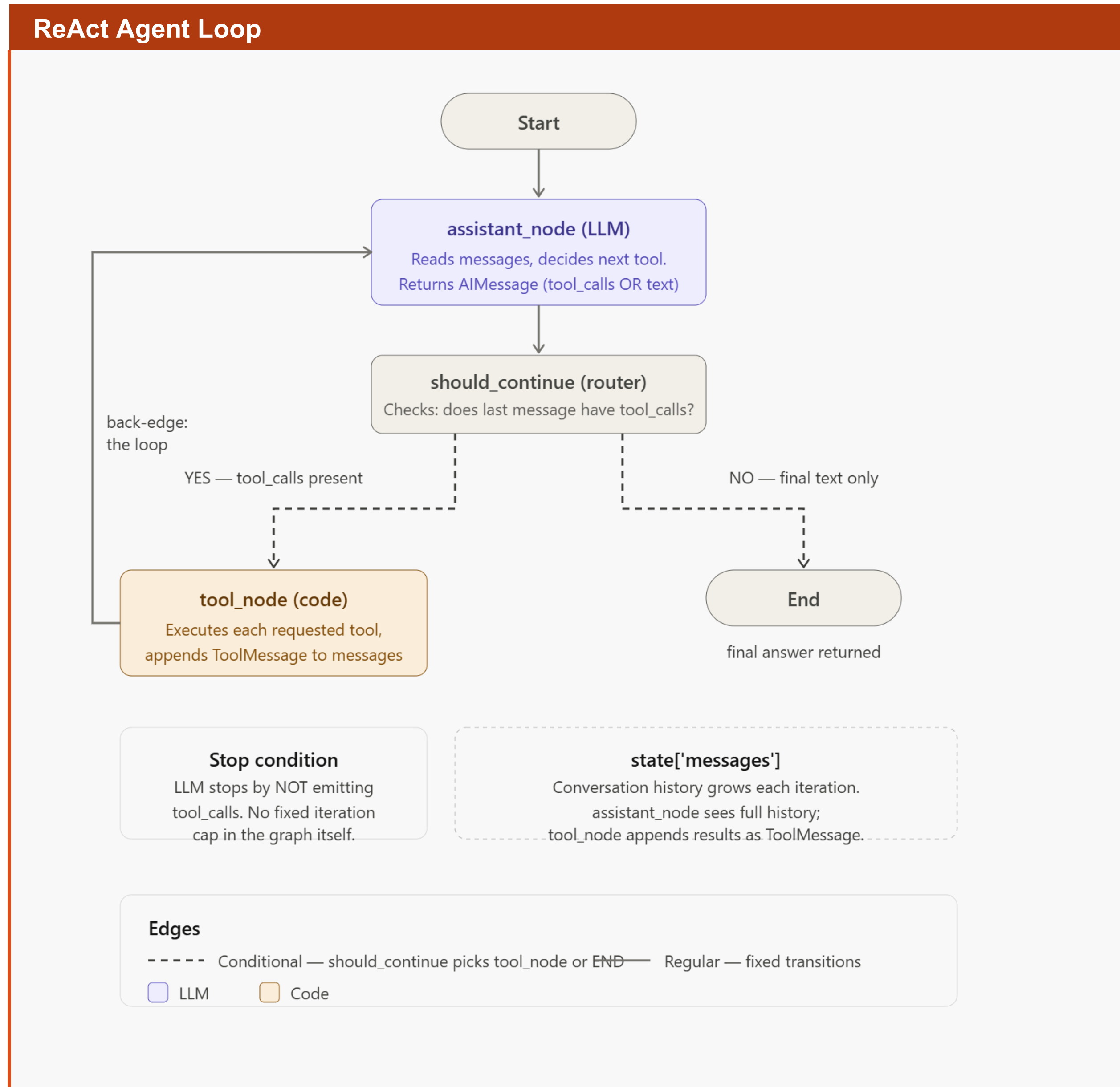
The dict in add_conditional_edges maps router return values to node names.
If your router returns 'fraud' -> map {'fraud': 'fraud_handler', ...}

Router functions should be fast and deterministic - no LLM calls inside them.

Pattern 3 - ReAct Agent (Reason and Act)

ReAct is the pattern used by all single agents. The agent alternates between Reasoning (LLM call) and Acting (tool execution) until it produces a final answer. The loop is a graph with a conditional back-edge.

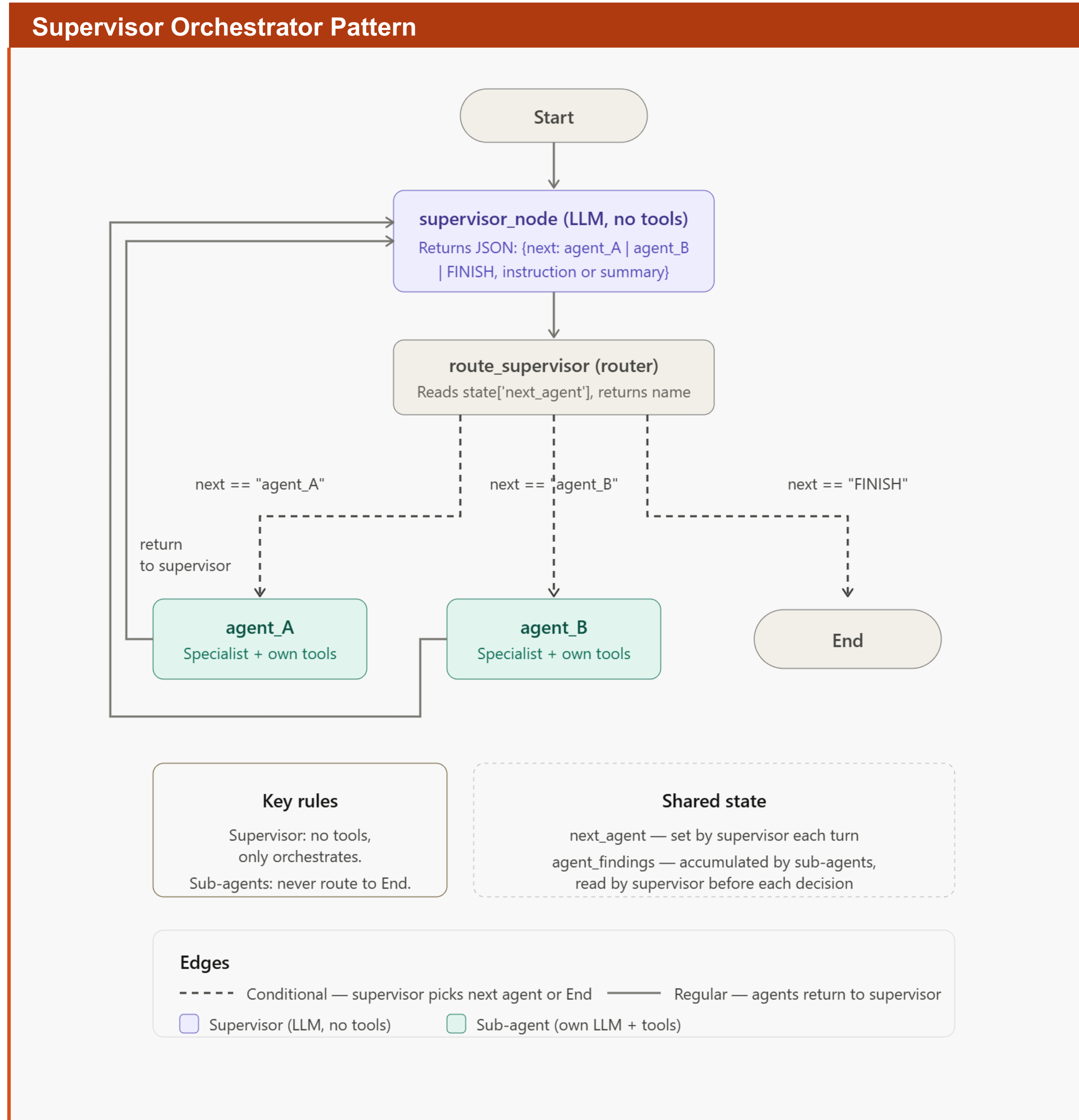
ReAct Agent Loop



The ToolNode in LangGraph is a pre-built node (from langgraph.prebuilt import ToolNode). You pass it the list of tool functions. It reads tool_calls from the last AIMessage, executes each tool by name, and appends ToolMessage results to the messages list. You never write ToolNode yourself.

Pattern 4 - Orchestrator (Supervisor)

The Supervisor pattern is used for multi-agent systems. The supervisor is a node (not a ToolNode) that reads accumulated findings from all sub-agents and decides which specialist to call next. It writes its routing decision as a string into state['next_agent'].



Pattern 5 - Reducer

A reducer is a function that controls how state field updates are merged. Instead of the default overwrite behaviour, a reducer lets you accumulate values across multiple node executions.

Reducer	Behaviour	Use Case
(none - default)	Last write wins - new value overwrites old	Simple scalar fields: counts, flags, strings
add_messages	New messages APPENDED to list, never replaced	Agent message history - preserves all tool calls
Custom reducer	You define the merge logic	Accumulating findings, merging dicts, summing counts

Why add_messages is Critical for Agents

Without add_messages:
  Node A returns {'messages': [msg1]} -> state.messages = [msg1]
  Node B returns {'messages': [msg2]} -> state.messages = [msg2]   msg1 is LOST!

With add_messages:
  Node A returns {'messages': [msg1]} -> state.messages = [msg1]
  Node B returns {'messages': [msg2]} -> state.messages = [msg1, msg2] preserved

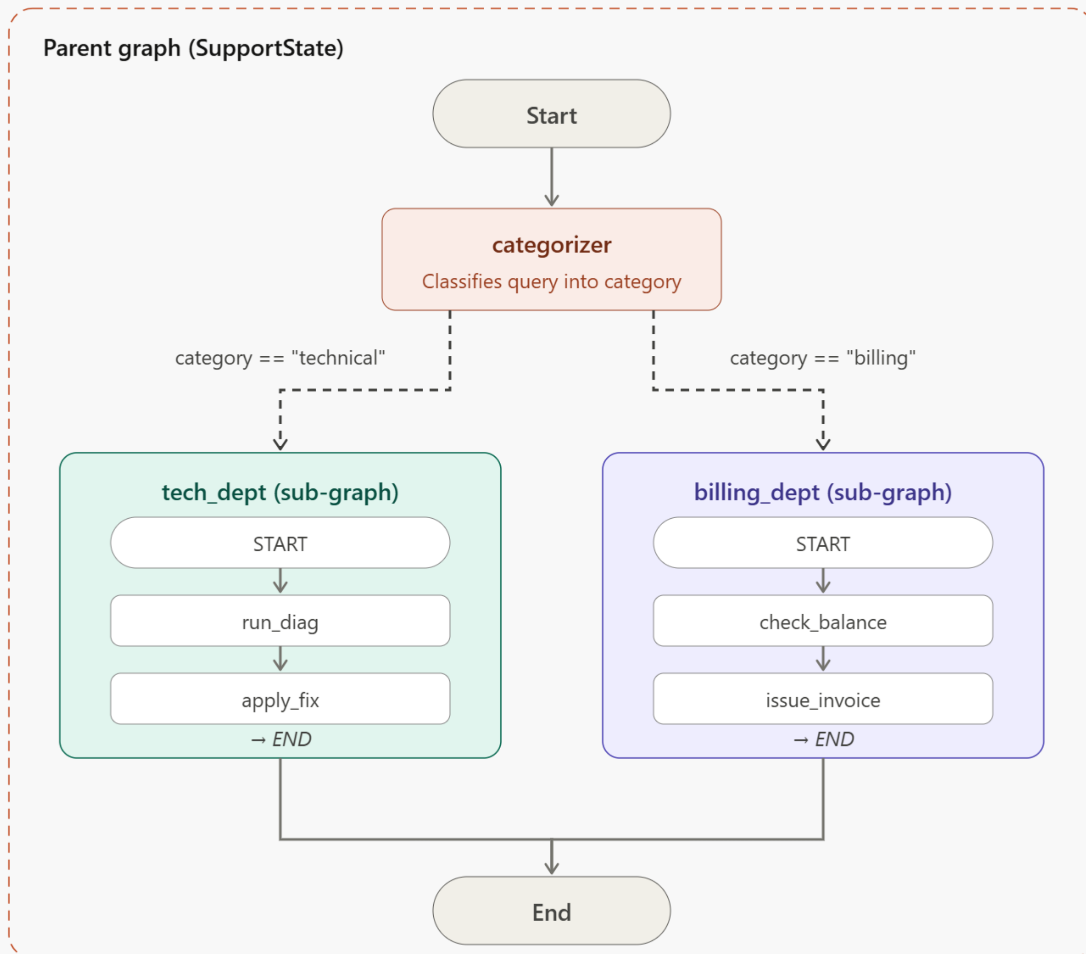
The agent loop requires the LLM to read ALL previous tool calls and results.
Without add_messages, the agent has no memory between iterations.

Pattern 6 - Sub-graphs

A sub-graph is a compiled LangGraph StateGraph embedded as a node inside a parent graph. Sub-graphs enable modular, reusable agent components - each sub-graph has its own state, nodes, and edges, but participates in the parent graph's execution.

Sub-graph Architecture

```
Parent Graph:
  START --> preprocessing --> [SUB-GRAPH NODE] --> postprocessing --> END

Sub-graph (compiled separately, used as a node in parent):
  START --> step_1 --> step_2 --> END

Code:
  sub = build_sub_graph()           # compiled StateGraph
  parent.add_node('my_sub', sub)    # used as a node in parent
  parent.add_edge('prev', 'my_sub')
  parent.add_edge('my_sub', 'next')

Use sub-graphs when a complex process is reused in multiple places,
or when a team owns a component independently from the rest.
```



Why use sub-graphs?

- **Modularity.** A complex workflow becomes one named node.
- **Reusability.** The same compiled sub-graph can be used in multiple parent graphs.
- **Team ownership.** Different teams own different sub-graphs and ship them independently.
- **State isolation.** Private fields stay private — the parent never sees them.
- **Independent testing.** Each sub-graph can be invoked, traced, and tested on its own.

How state flows across the boundary

The parent and the sub-graph each have their own TypedDict. LangGraph passes state across the boundary by field-name overlap: shared fields flow in and merge out, sub-graph-only fields are private.

Mental Model — Shared Fields are the Interface

Think of overlapping field names as the function signature of the sub-graph. If the parent has user_id, query, and resolution_steps, and the sub-graph declares query and resolution_steps, the contract is: "give me a query, I will append to resolution_steps." Anything else the sub-graph computes lives only inside it.

Building and wiring — three steps

```
# 1. Build sub-graph with its own state, then COMPILE it
tech_subgraph = tech_builder.compile()

# 2. Use the compiled sub-graph as a NODE in the parent
parent.add_node('tech_dept', tech_subgraph)
parent.add_edge('categorizer', 'tech_dept')
```

Common pitfalls

- **Forgetting to compile.** You must call `.compile()` before adding the sub-graph to the parent.
- **Schema mismatch on shared fields.** If the parent says `List[str]` and the sub-graph says `Annotated[List[str], add]`, reducers will not align. Keep types and reducers identical across boundaries.
- **Visualisation hides internals by default.** Pass `xray=1` to `get_graph()` to expand sub-graph internals in the rendered diagram.
- **Over-nesting.** Two levels of nesting is comfortable, three is the limit before debugging gets painful.

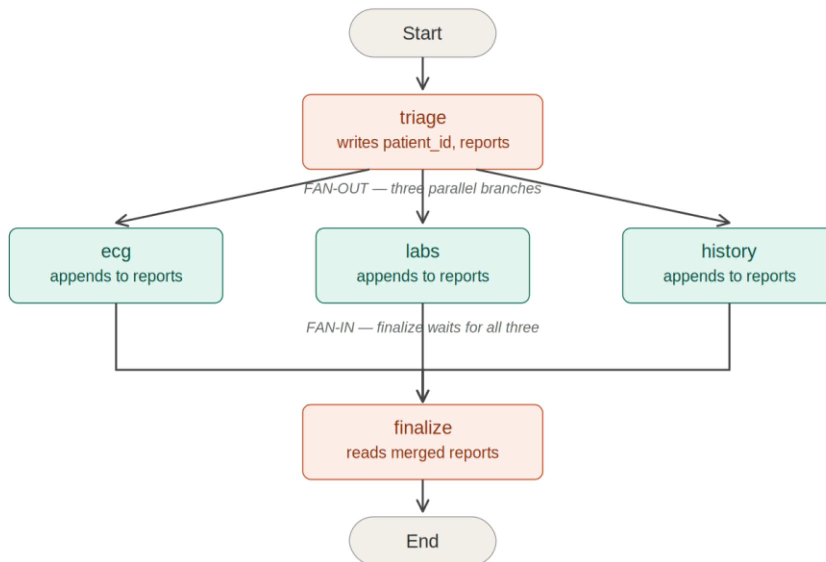
Sub-graph or Multi-Agent Supervisor?

Use a sub-graph when routing is fixed at design time ("after categorizer, go to tech_dept or billing_dept based on a label"). Use a supervisor when routing requires the LLM to read accumulated findings and decide what to do next. The two compose: a supervisor's sub-agents are typically themselves sub-graphs.

Pattern 7 – Parallelization (Fan-out / Fan-in)

Many agent tasks contain steps that do not depend on each other. A diagnostic agent may need to fetch ECG data, lab results, and medical history before forming an opinion — but none of those lookups depends on the others. Running them sequentially is wasted wall-clock time.

LangGraph lets you fan-out from one node to several nodes that execute in parallel, then fan-in to a single node that consumes all their results.



Hospital Triage — fan-out to three parallel branches, fan-in at finalize

The Cost of Sequential Execution

If three tool calls each take 2 seconds and you run them in order, the user waits 6 seconds. Run them in parallel and the user waits 2 seconds — the slowest of the three.

The two halves of every parallel pattern

- **Fan-out:** call `add_edge(source, target)` once per parallel target. LangGraph schedules them concurrently.
- **Fan-in:** call `add_edge([n1, n2, n3], "aggregator")` — pass a list of source nodes to create a sync point that waits for all of them.

The race condition problem

Parallel nodes that all write to the same state field create a race condition. By default, LangGraph uses last-write-wins for state merges. When two parallel nodes write the same field, LangGraph cannot decide which write is authoritative and raises InvalidUpdateError.

Why Last-Write-Wins Breaks Parallel Writes

Sequential nodes are safe — Node A writes, then Node B overwrites (usually the intent). Parallel nodes write in the same step. If Node B and Node C both return {state: ['B']} and {state: ['C']}, LangGraph crashes rather than silently picking one — which is the right behaviour, because silent data loss in production would be far worse.

Reducers — the fix for parallel writes

A reducer is a function attached to a state field that tells LangGraph how to merge updates. Declare it with `Annotated[type, reducer_fn]` in the TypedDict. The most common reducer is `operator.add`, which appends list items instead of replacing the list.

```
from typing import Annotated, List
import operator

class State(TypedDict):
    reports: Annotated[List[str], operator.add]   # parallel-safe
```

Reducer reference

Reducer	Behaviour	When to use
(none — default)	Last write wins. Crashes if two parallel writes target the same field.	Single-writer fields: scalars, flags, classification labels.
`operator.add`	Concatenates lists. Each parallel write appends; nothing is lost.	Accumulating findings, retrieved chunks, agent reports.
`add_messages`	Appends messages with deduplication and message-ID handling.	Chat history in agent loops (ReAct, supervisor).
Custom function	You define merge logic. Signature: (existing, update) -> merged.	Summing counts, deep-merging dicts, conflict resolution.

Common pitfalls

- **Forgetting the reducer.** If any parallel node writes to a field, that field needs a reducer. The `InvalidUpdateError` only appears at runtime, not compile time.
- **Forgetting fan-in.** Without a list-form add_edge, the downstream node may run as soon as any one branch finishes. Always use `add_edge([...], "next")` to create a sync point.

- **Hidden coupling.** Branches you think are independent may secretly read state written by a sibling. Audit each parallel node's reads.
- **Cost amplification.** Three parallel LLM calls cost three times one. Parallelization saves wall-clock time, not money.

When NOT to Parallelize

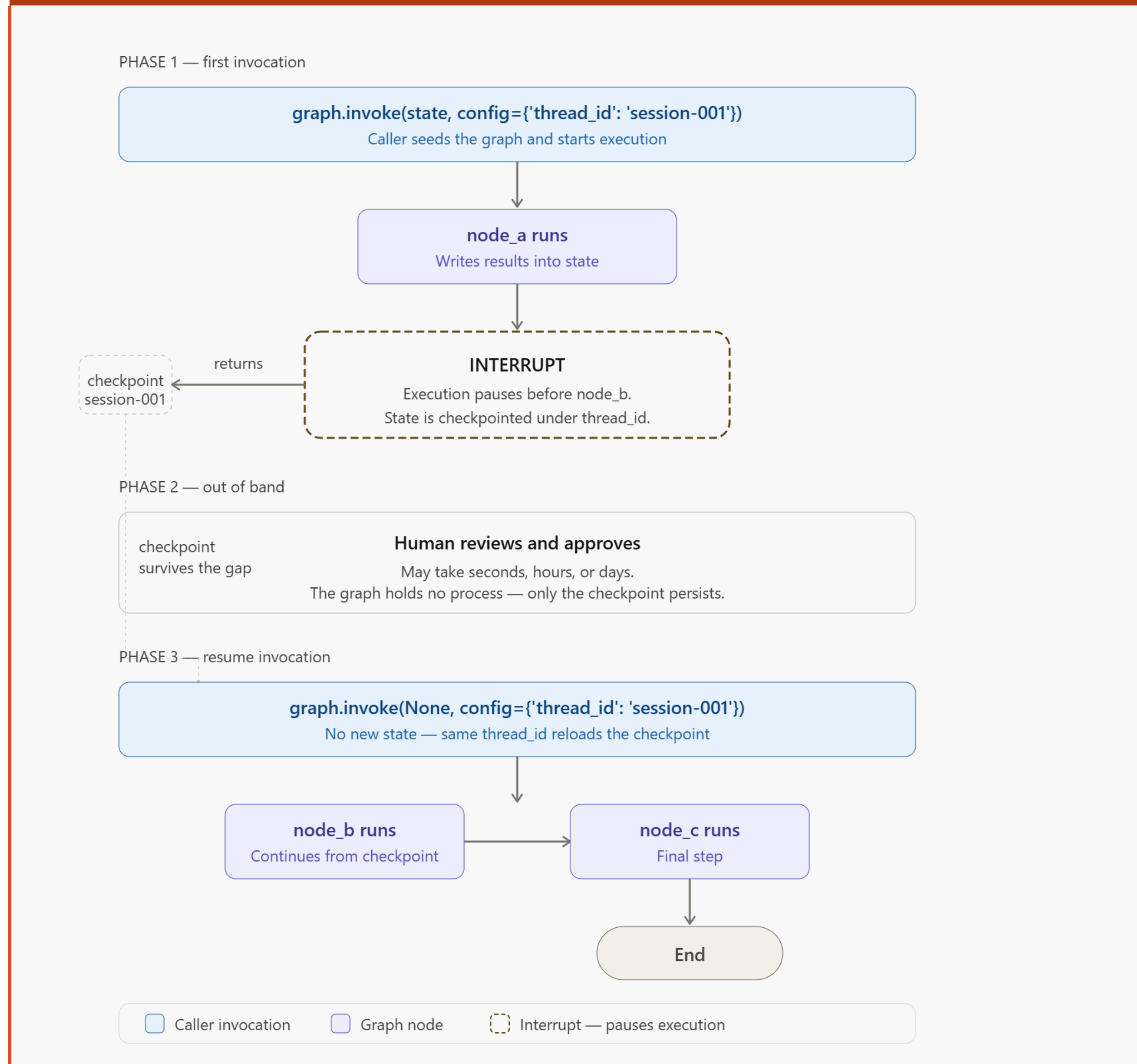
If Step B needs Step A's output, they cannot run in parallel — that is a sequential dependency. The acid test: could you run them in either order, in sequence, and get the same answer? If yes, they can run in parallel.

5.3 Human-in-the-Loop

LangGraph supports interrupting graph execution at any node so a human can review, approve, or modify the state before the graph continues. This is essential for production systems where an agent might take consequential actions.

- **interrupt_before:** pause BEFORE a node executes - let the human approve the planned action
- **interrupt_after:** pause AFTER a node executes - let the human review the result before continuing
- **Checkpointer:** required for human-in-the-loop - saves state to a database (SQLite, Postgres)

Human-in-the-Loop Flow



5.4 The Complete State Lifecycle

- **invoke() called** - initial state dict passed in (question, empty message list, etc.)
- **START node** - LangGraph routes to the first registered node
- **Node executes** - reads state, does work, returns a partial dict
- **State merge** - LangGraph applies reducers and merges returned dict into state
- **Edge evaluated** - static edge or conditional edge decides next node
- **Steps 3 to 5 repeat** - until the graph reaches END
- **invoke() returns** - the final state dict (read any field from it)

Chapter 6

MCP - Model Context Protocol

Connecting agents to the real world

6.1 What is MCP?

MCP (Model Context Protocol) is an open standard introduced by Anthropic in 2024. It defines a universal interface that allows AI agents to connect to external data sources and tools without each agent developer writing custom integration code for every service.

Before MCP, connecting an agent to an external service required:

- **Writing a custom @tool function per service**
- **Handling authentication for each service separately**
- **Managing the API client lifecycle inside the tool**
- **Repeating this entire process for every new service**

With MCP, an MCP Server handles all of this. The agent connects to the MCP Server using a standard protocol and automatically gets access to all the tools the server exposes.

MCP Architecture

```
LangGraph Agent
      |
      |  (standard MCP protocol - HTTP/SSE or stdio)
      |
      v
MCP Server --> External Service (Gmail, Google Drive, Slack, Jira, ...)

The agent does NOT need to know how Gmail works.
The MCP Server handles: authentication, API calls, error handling.
The agent just calls the tool by name - the server executes it.

Examples of MCP Servers (all open source):
  gmail, google-calendar, slack, jira, asana, github, postgres, ...
```

6.2 MCP Components

Component	Role	Who Creates It
MCP Client	The agent - initiates connection, calls tools	You (your LangGraph agent code)

Component	Role	Who Creates It
MCP Server	Exposes tools over the standard protocol	Service providers or open source community
Tool	A callable function the server exposes	Defined by the MCP Server developer
Resource	Data the server can provide (files, records)	Defined by the MCP Server developer
Prompt	Pre-built prompt templates from the server	Defined by the MCP Server developer

6.3 MCP vs Custom @tool Functions

Aspect	Custom @tool	MCP Server
Where code runs	Inside your agent process	Separate MCP Server process
Authentication	You handle it in the tool	MCP Server handles it
Reusability	Only for your agent	Any agent can connect to the same server
Maintenance	You maintain the integration	Server provider maintains it
Tool discovery	You hardcode the tool list	Agent discovers tools automatically from server
Best for	Custom business logic, private data	Standard services: Gmail, Jira, GitHub, Slack

MCP Server Configuration in LangGraph

```
mcp_servers = [
    {
        'type': 'url',
        'url':  'https://gmail.mcp.claude.com/mcp',
        'name': 'gmail-mcp'
    }
]

# Agent automatically discovers all tools the server exposes.
# No custom @tool functions needed for Gmail operations.
# Agent can call list_emails(), send_email(), search_inbox() etc.
# - all discovered automatically from the MCP Server.
```

Chapter 7

Observability and Langfuse

Monitor, debug, and optimise every agent run

7.1 Why Observability Matters for Agents

Traditional software is deterministic: the same input always produces the same output via the same code path. You can unit-test it and be confident it works in production.

Agentic AI is non-deterministic: the LLM makes different decisions on different runs. An agent that passed testing may behave unexpectedly in production. Without observability, you cannot know:

- **Which tools the agent called and in what order**
- **What exact prompt was sent to the LLM at each step**
- **Where in the graph the wrong decision was made**
- **How many tokens were used and what cost was incurred**
- **Whether the agent is getting faster or slower over time**

Observability gives you visibility into the agent's reasoning process - every LLM call, every tool execution, every routing decision - exactly as it happened in production. This is not optional for production agentic systems; it is essential.

7.2 Observability Tools

Tool	Type	Strengths	Best For
Langfuse	Open source / Cloud	LangGraph graph visualisation, sessions, scores	Production LangGraph agents - our main tool
LangSmith	Commercial (LangChain)	Deep LangChain integration, datasets, evaluation	Teams already in the LangChain ecosystem
Arize Phoenix	Open source	ML observability, LLM evals, free self-hosted	Teams wanting full self-hosted control
Helicone	Commercial SaaS	Proxy-based - no SDK changes needed, cost tracking	Quick cost visibility with minimal setup
W&B (Weights & Biases)	Commercial	Experiment tracking, model comparison, dashboards	ML training plus agent experiments
OpenTelemetry	Open standard	Universal observability, works with any backend	Large engineering teams with existing OTel

Tool	Type	Strengths	Best For
PromptLayer	Commercial	Prompt versioning, A/B testing, usage analytics	Prompt engineering and iteration workflows

7.3 Langfuse Deep Dive

Langfuse is the observability tool used throughout this course. It is open source (MIT licence), can be self-hosted, and has native support for LangGraph - meaning it shows agent runs as actual graph visualisations, not flat log lists. Each compiled LangGraph StateGraph invoked with the Langfuse callback handler appears as a fully visualised trace.

Key Concepts in Langfuse

Concept	What It Is	Equivalent In LangGraph
Trace	One complete agent run from start to finish	One graph.invoke() call
Span	One step within a trace	One node execution or LLM call
Generation	One LLM call with input/output/tokens	One get_llm().invoke() call
Session	Group of traces for the same user/conversation	Set by langfuse_session_id in metadata
Score	Human or automated quality rating on a trace	Added manually or via eval pipeline

The Rule for Graph Visualisation in Langfuse

For Langfuse to show a GRAPH VISUALISATION of your agent:
1. The agent must be a compiled LangGraph StateGraph (builder.compile())
2. The invoke() call must pass config with the Langfuse callback handler
3. Each invoke() appears as a separate trace in Langfuse

Simple RAG trace: __start__ -> rag_search -> rag_simple -> __end__
Agentic RAG trace: __start__ -> rag_search -> rag_reason -> rag_search ->
rag_reason -> rag_answer -> __end__ (loop is visible!)

A plain Python loop calling llm.invoke() directly shows as a FLAT SPAN - no graph.

The Langfuse Setup Pattern

utils/langfuse_setup.py - The Standard Pattern

```
from langfuse import Langfuse, get_client
from langfuse.langchain import CallbackHandler
```

```
def init_langfuse():
    langfuse = Langfuse()          # reads LANGFUSE_* vars from .env
    handler  = CallbackHandler()   # pass to every graph.invoke()
    return langfuse, handler

def make_config(handler, run_name, tags):
    return {
        'callbacks': [handler],
        'run_name':  run_name,
        'metadata': {
            'langfuse_session_id': 'my-session',
            'langfuse_tags':       tags,
        }
    }

# Usage in main.py:
lf, handler = init_langfuse()
result = graph.invoke(state, config=make_config(handler, 'agent-run',
['rag']))
get_client().flush()  # send all buffered traces before exit
```

Setting Up Langfuse

- Create an account at **cloud.langfuse.com** (free tier available)
- Get Public Key and Secret Key from the project settings
- Add to your .env: LANGFUSE_PUBLIC_KEY, LANGFUSE_SECRET_KEY, LANGFUSE_HOST
- Initialise: langfuse, handler = init_langfuse()
- Pass to every graph.invoke(): config=make_config(handler, run_name, tags)
- Call get_client().flush() before your program exits

Reading a Langfuse Trace

- **Timeline view:** horizontal bars showing each node/LLM call in chronological order
- **Graph view:** your LangGraph nodes and edges visualised - the loop is visible
- **Input/Output:** exact prompt sent to LLM and exact response received at each step
- **Token counts:** input tokens, output tokens, cost per LLM call and per trace
- **Latency:** time taken per node - identifies bottlenecks
- **Sessions:** group all traces for one conversation - see patterns across turns

7.4 What Good Observability Tells You

Question	Where to Look in Langfuse
Why did the agent give a wrong answer?	Trace -> find node where reasoning went wrong -> read the exact prompt
Is the agentic loop working correctly?	Graph view -> count how many times rag_reason ran
Which node is slowest?	Timeline view -> look for the widest bar
How much is one run costing?	Trace header -> total tokens and cost
Is quality improving over time?	Scores tab -> add manual scores or automate with evals
Are users satisfied?	Sessions -> group traces by user ID and look for patterns

Chapter --

Quick Reference

Key facts you should know from memory

Everything in One Table

Concept	Key Fact to Remember
LangChain vs LangGraph	LangChain = pipeline. LangGraph = state machine. LangGraph is built ON LangChain.
Workflow vs Agent	Workflow: developer decides steps. Agent: LLM decides steps at runtime.
Single Agent vs Multi-Agent	Single: one LLM + tools. Multi: supervisor + specialists, each with own tools.
add_messages	Appends to message list, never overwrites. Without it, agents lose memory between iterations.
Supervisor rule	Supervisor has NO tools. Only orchestrates. Only supervisor can route to END.
agent_findings	Dict in state where each sub-agent writes its findings. Supervisor reads all before FINISH.
Traditional RAG	One fixed search with original question. One LLM call. Cannot adapt to what it finds.
Agentic RAG	LLM decides search queries. Multiple searches. Loop until LLM is confident enough.
MCP	Standard protocol for agent-to-service connections. Agent discovers tools automatically from server.
Langfuse graph view	Only shows for compiled LangGraph StateGraph with callbacks in invoke() config.
ReAct pattern	Reason -> Act -> Reason -> Act loop. tool_calls present? -> ToolNode -> repeat.
should_continue	Checks last AIMessage for tool_calls. YES -> tool_node. NO -> END.
route_supervisor	Reads state['next_agent'] written by supervisor LLM. Returns next node name.
Checkpointing	Required for human-in-the-loop. Saves state so graph can be resumed after interrupt.
Star topology	All sub-agents have static edges back to supervisor. No agent goes directly to END.
CrewAI limitation	Can only use pre-defined agent patterns. Cannot build custom graph architectures.

Concept	Key Fact to Remember
ToolNode	Pre-built LangGraph node. Reads tool_calls from last AIMessage, executes them, returns ToolMessages.
Reducer	Function controlling how state updates merge. Default: overwrite. add_messages: append.

Glossary

Term	Definition
LCEL	LangChain Expression Language - the pipe operator syntax for chaining LangChain components
TypedDict	Python dict with type annotations - used as LangGraph state definition
Reducer	Function controlling how state field updates are merged (default: overwrite, add_messages: append)
ToolNode	Pre-built LangGraph node that executes tool calls from the last AIMessage in state
StateGraph	LangGraph's graph builder class - defines nodes, edges, and state schema
Compiled graph	Result of builder.compile() - what you actually call invoke() on
Thread ID	Identifier for a conversation - used with checkpointer for human-in-the-loop
ReAct	Reason and Act - prompting pattern where LLM alternates reasoning and tool calls
FAISS	Facebook AI Similarity Search - in-process vector database for fast semantic search
Embedding	Dense vector representation of text - used for semantic similarity search in RAG
MCP	Model Context Protocol - Anthropic open standard for agent-to-service connections
Langfuse	Open-source LLM observability with native LangGraph graph visualisation
next_agent	State field written by supervisor to signal which sub-agent runs next
agent_findings	State field accumulating each sub-agent's report for the supervisor to read
Star topology	Multi-agent graph shape - all sub-agents connect back to the supervisor hub
Sub-graph	Compiled StateGraph used as a node inside a parent graph - enables modular design
Checkpoint	Saved state snapshot - enables human-in-the-loop and fault-tolerant long runs
Interrupt	LangGraph mechanism to pause before or after a node for human review