GRAPH ENGINEERING FOR AI AGENTS

BUILDING RELIABLE, SCALABLE AGENT SYSTEMS WITH COMPUTATIONAL GRAPHS

FROM IDEA TO IMPACT WITH GRAPH-POWERED AI AGENTS

A Practical Guide to Design, Build, Orchestrate, and Operate Agent Systems that Scale

GRAPH MODELING & DATA STRUCTURES

AGENT LOGIC, TOOLS & WORKFLOWS
LANGGRAPH, LLMs, FUNCTIONS & MORE

STATE MANAGEMENT, MEMORY & CONTEXT

SCALABILITY, RELIABILITY & FAULT TOLERANCE

OBSERVABILITY, EVALUATION & CONTINUOUS IMPROVEMENT

DESIGN • BUILD • DEPLOY • SCALE • SECURE • MAINTAIN

GRAPH SYSTEMS IN ANY ENVIRONMENT

BRENT TAYLOR

Graph Engineering for AI Agents

Building Reliable, Scalable Agent Systems with Computational Graphs

Steve Publications

This book is available at https://leanpub.com/graphengineeringforaiagents

This version was published on 2026-07-30

Contents

Building Reliable, Scalable Agent Systems with Computational Graphs

About this book: AI agents are no longer simple prompt-and-response wrappers around language models. Production-grade agents plan, use tools, collaborate with other agents, remember past interactions, recover from failures, and handle human oversight. None of this can be expressed cleanly as a linear pipeline. This book teaches you to model agent systems as computational graphs – directed structures of nodes and edges that encode state, control flow, parallelism, and recovery. Through detailed explanations, complete code examples, and real-world architectures, you will learn to design, implement, debug, scale, and secure graph-based AI agents using modern frameworks like LangGraph. By the end, you will have the knowledge to build agent systems that are reliable under failure, observable in production, and capable of the complex behaviors that define next-generation AI applications.

Introduction: Why Graphs for AI Agents

The Linear Pipeline Problem

Consider a customer support agent. It receives an email, classifies the intent, looks up account information, drafts a response, asks a human to review it, incorporates feedback, and sends the final reply. On paper, this looks like a sequence. In practice, it is not.

The classification might be uncertain, requiring the agent to search for more context before deciding. The account lookup might fail, demanding a retry or a fallback strategy. The drafted response might need multiple rounds of human review. The user might interrupt to change direction mid-flow. Any of these paths could loop back to an earlier step, branch into parallel subtasks, or terminate early.

A linear pipeline – the chain of function calls that dominated early LLM application development – cannot express this behavior natively. You can bolt on conditionals, try-except blocks, and state variables, but the result is spaghetti code that grows exponentially harder to maintain with each added edge case. The fundamental problem is architectural: a sequence of steps assumes a world where execution proceeds predictably from start to finish. AI agents do not live in that world.

The linear approach worked when LLM applications were simple: prompt, generate, return. But as agents grew more capable, the limitations became acute. Teams building production systems at companies like Uber, Klarna, and J.P. Morgan hit a wall around late 2023 and early 2024. Their agents needed to maintain state across hours or days, recover from infrastructure failures without losing progress, coordinate multiple specialized sub-agents, and incorporate human review at arbitrary points in their workflows. Linear chains could not deliver this reliably.

Consider Klarna's AI Assistant, which handles customer support for 85 million active users [29,30]. The assistant must classify intent, look up account

data, draft responses, route to human agents when confidence is low, and maintain conversation context across sessions that span days. A single linear chain cannot express the branching logic needed: what happens when the account lookup fails? When the drafted response requires legal review? When the user switches topics mid-conversation? Klarna's solution uses LangGraph to model these paths as a stateful directed graph with explicit error handling, human-in-the-loop checkpoints, and durable execution. The result: an 80 percent reduction in customer resolution time and automation of approximately two-thirds of all customer inquiries [30,48].

Uber's Developer Platform team faced a different challenge: automating unit test generation across thousands of codebases. Their agent network must understand code context, generate tests, execute them, analyze failures, and iterate until tests pass. This is inherently cyclic, with each failed test informing the next generation attempt. Uber built this as a LangGraph-based workflow that saved approximately 21,000 developer hours [29,47]. The graph topology makes the generate-test-fix loop explicit, with checkpointing ensuring that long-running test suites can resume from interruption.

J.P. Morgan deployed LangGraph agents for financial analysis workflows that require strict audit trails, deterministic routing, and human approval gates at critical decision points. Their agents must maintain state across multi-day research sessions, recover from infrastructure failures without losing analytical progress, and produce auditable execution paths that regulators can review. Graph-based architecture provides all of these capabilities natively: checkpointing for durability, explicit edges for auditability, and interrupt-based pauses for human oversight [29,30].

These production deployments share a common architectural insight: complex agent behavior cannot be expressed as a sequence. It requires the branching, looping, stateful, and recoverable execution that only graph-based models provide.

The solution was to treat agent behavior as a graph problem from the start.

Graphs as Cognitive Architecture

A directed graph is a collection of nodes connected by edges that indicate relationships or flow. In the context of AI agents, nodes represent computational steps – an LLM call, a database query, a tool execution, a human review

checkpoint. Edges represent the rules that determine which step follows which, including conditional routing based on state, parallel branching for concurrent work, and cycles for iterative refinement.

This model maps naturally to how intelligent systems actually operate. Human cognition is not linear. When you solve a problem, you explore possibilities, backtrack when you hit dead ends, consult references, verify intermediate results, and iterate toward a solution. Graphs capture this non-linear, stateful, branching behavior natively.

The connection between graphs and agent architecture is not new in the broader computing world. Dataflow programming, dating back to Dennis and Barnes in the 1970s, represents programs as directed graphs where nodes are operations and edges carry data tokens. Execution proceeds when all inputs to a node become available, enabling natural parallelism. TensorFlow popularized computational graphs for machine learning inference, where each node is a mathematical operation and edges represent tensor flow. Apache Beam and Google's Pregel system use graph-based execution models for large-scale data processing, operating in discrete "super-steps" with synchronization barriers between them.

What makes graph engineering specifically important for AI agents is the combination of three factors that distinguish agents from traditional data processing pipelines:

First, AI agents are inherently stateful. They maintain conversation history, accumulated knowledge, tool call results, and intermediate reasoning across multiple steps. The state evolves as nodes execute, and each node reads and writes to this shared context. Graph-based execution models provide explicit state management with well-defined update semantics, including reducers that merge concurrent updates safely.

Second, AI agents require non-deterministic control flow. Unlike a data processing pipeline that always follows the same path, an agent's next step depends on LLM outputs, tool results, and external conditions. Conditional edges in a graph encode this branching explicitly, making the decision logic visible and debuggable rather than hidden inside if-else chains.

Third, AI agents need cycles. An agent might iterate between generating a response, evaluating its quality, and refining it until a threshold is met. It might loop through tool calls, each informing the next. Linear DAGs (directed acyclic graphs) cannot express loops; general directed graphs can, and modern graph

execution engines like LangGraph support them natively with configurable recursion limits.

From DAGs to Dynamic Execution

Traditional workflow orchestration tools like Apache Airflow and early versions of Prefect enforce acyclicity by design. They model workflows as DAGs, which works perfectly for ETL pipelines and batch processing but fails for agent systems that need loops, dynamic branching, and runtime decision-making.

Graph-based agent frameworks break this constraint. LangGraph, released in early 2024, was built specifically to address the limitations of linear chains and acyclic workflows. It models agents as stateful directed graphs that can contain cycles, with an execution engine inspired by Pregel's super-step model. Each super-step executes all currently eligible nodes, merges their state updates, and proceeds to the next step until no nodes remain active.

This execution model provides several critical capabilities:

Durable execution. The graph checkpoints state after each node completion, enabling crash recovery. If a process dies mid-execution, it can resume from the last checkpoint without losing progress. This is essential for long-running agent workflows that span minutes or hours.

Human-in-the-loop. Execution can pause at any node, save state, and wait indefinitely for external input. When resumed, execution continues from exactly where it stopped, with the human's input injected as the return value of the paused operation.

Dynamic parallelism. Nodes that have no data dependencies on each other execute concurrently within a super-step. The Send API enables dynamic fan-out, spawning parallel worker tasks at runtime based on state rather than predefined topology.

Subgraph composition. A graph can be used as a node within another graph, enabling modular design. Complex multi-agent systems are built by composing specialized subgraphs, each encapsulating a specific capability, into an orchestration graph that coordinates their interaction.

What You Will Build

This book is structured to take you from first principles to production architectures. The early chapters establish the theoretical foundations and core mechanics of graph-based agent systems: what computational graphs are, how state flows through them, how to build nodes and edges, and how routing decisions work.

The middle chapters dive into the capabilities that make graphs essential for agents. You will learn to implement tool orchestration with error recovery, build reasoning patterns like chain-of-thought and self-correction loops, design multi-agent collaboration architectures, and integrate retrieval-augmented generation into graph workflows.

The later chapters address production concerns: streaming and async execution for responsive interfaces, memory systems that enable agents to learn across sessions, debugging and observability tools, performance optimization, fault tolerance, security guardrails, and deployment patterns for scaling to thousands of concurrent users.

Throughout the book, you will work with complete, runnable code examples using LangGraph as the primary framework. The concepts transfer to other graph-based frameworks, but LangGraph provides the most mature and well-documented implementation of graph-based agent orchestration, trusted by companies running agents in production. Each code example is designed to compile and run without modification, and each concept is explained with enough depth that you understand not just how to use it, but why it works the way it does.

By the end of this book, you will be able to design and implement AI agent systems that are more than prototypes. You will build agents that handle failures gracefully, scale under load, incorporate human oversight naturally, and exhibit the complex, adaptive behaviors that define truly intelligent systems.

Chapter 1: Foundations of Computational Graphs

Graph Theory Refresher for Engineers

A directed graph is a mathematical structure consisting of vertices (nodes) and directed edges that connect pairs of vertices. Formally, a directed graph G is defined as G = (V, E), where V is a set of vertices and E is a set of ordered pairs (u, v) representing directed edges from vertex u to vertex v.

For our purposes, we need only a few concepts from graph theory, and they are straightforward:

- **Node (vertex):** A unit of computation. In agent graphs, a node is typically a Python function that reads shared state, performs some operation, and returns a partial state update.
- **Edge:** A directed connection between nodes that defines execution flow. An edge from node A to node B means that after A completes, B may execute next.
- **Path:** A sequence of connected edges from one node to another. The set of all possible paths through a graph represents the agent's behavioral space.
- **Cycle:** A path that starts and ends at the same node. Cycles enable loops, which are essential for iterative reasoning and retry logic.
- **DAG (Directed Acyclic Graph):** A directed graph with no cycles. DAGs are useful for simple sequential workflows but cannot express the looping behavior agents need.
- **In-degree / Out-degree:** The number of edges entering or leaving a node. A node with in-degree zero has no predecessors (a starting point). A node with out-degree zero is a terminal node.

These concepts are not abstract mathematics. They are the vocabulary you will use daily when designing agent systems. When you add a conditional edge to handle an error case, you are modifying the graph's topology. When you

create a loop for self-correction, you are introducing a cycle. When you fan out to parallel workers, you are increasing the out-degree of a routing node.

The distinction between acyclic and cyclic graphs is particularly important. Traditional workflow systems like Apache Airflow enforce DAGs because cycles in batch processing pipelines usually indicate bugs – infinite loops that never terminate. Agent systems are different. Cycles are not bugs; they are features. An agent that generates code, tests it, observes the failure, and tries again is executing a cycle. This loop continues until the code passes or a maximum iteration count is reached.

Figure 1.1: Acyclic vs Cyclic Agent Graphs

This figure illustrates the fundamental topological difference between DAG-based workflows and cyclic agent graphs. The left panel shows a traditional DAG with three sequential nodes (A -> B -> C) and one conditional branch, all terminating at END. There is no path that returns to an earlier node. The right panel shows a cyclic agent graph where the "evaluate" node has edges back to "generate" (for retry), to "fallback" (for error recovery), and to END (for success). The cycle allows the agent to iterate until quality thresholds are met. Key annotation: cycles are controlled by recursion limits and explicit termination conditions in the routing logic, preventing infinite loops while enabling necessary iteration.

Topology comparison:

- **DAG:** Nodes {START, A, B, C, END}, Edges {(START,A), (A,B), (B,C), (C,END)}. Maximum path length: 3 edges. Guaranteed termination.
- **Cyclic graph:** Nodes {START, generate, test, evaluate, fallback, END}. Edges include (evaluate, generate) creating a cycle. Path length is variable, bounded by recursion limit. Termination controlled by routing function.

Understanding this distinction is not academic. It determines what your agent can do. If you need iteration – retrying failed operations, refining answers based on self-evaluation, exploring multiple reasoning paths – you need cycles. If your workflow is purely sequential with optional branching, a DAG suffices. Most production agent systems use both: DAGs for deterministic preprocessing and postprocessing, cyclic graphs for the core reasoning and tool-use loops.

Types of Execution Graphs

Not all graphs used in AI systems serve the same purpose. Understanding the distinctions between graph types helps you choose the right model for each problem:

Computational graphs represent programs as networks of operations connected by data dependencies. Each node performs a computation, and edges indicate which node's output feeds into another node's input. TensorFlow popularized this model for machine learning, where each node is a mathematical operation (matrix multiplication, activation function, etc.) and edges carry tensors. In agent systems, computational graphs express the flow of data between processing steps.

Execution graphs (also called control-flow graphs) represent the sequence and branching of program execution. Unlike pure computational graphs that focus on data flow, execution graphs emphasize the order and conditions under which operations run. Agent frameworks like LangGraph build execution graphs where nodes are functions and edges encode the routing logic – static, conditional, or dynamic.

Workflow graphs model business processes or multi-step procedures as sequences of tasks with dependencies. They are typically DAGs designed for batch processing, where each task must complete before dependent tasks begin. Workflow graphs are useful for the deterministic portions of agent systems, such as data preparation steps that run before the agent begins reasoning.

State graphs combine computation and state management into a single model. Each node reads and writes to a shared state object, and edges determine the next node based on current or updated state. LangGraph's core abstraction is a StateGraph, where the state schema defines what data flows through the graph, nodes update that state, and edges route execution.

Dependency graphs encode prerequisites between tasks. If task B depends on the output of task A, there is an edge from A to B. Dependency graphs are implicit in most agent frameworks – they determine which nodes can execute in parallel (no dependency) versus which must wait (dependency exists).

Reasoning graphs model the structure of thought processes. Chain-of-thought reasoning maps to a linear path of nodes. Tree-of-thoughts creates

branching paths where multiple candidate solutions are explored. Graph-of-thought allows arbitrary connections between reasoning steps, enabling cross-attention between different branches of analysis.

Orchestration graphs coordinate multiple agents or services. They sit at a higher level than individual agent execution graphs, managing the lifecycle, communication, and resource allocation of multiple agent instances working toward a shared goal.

Multi-agent interaction graphs model how agents communicate with each other. Nodes represent agents, and edges represent communication channels – direct messages, shared state updates, or broadcast announcements. These graphs can be static (fixed team structure) or dynamic (agents form and dissolve collaborations at runtime).

In practice, a production agent system layers several of these graph types. An orchestration graph coordinates multiple agents, each of which executes its own state graph containing reasoning nodes, tool execution nodes, and memory management nodes. The interaction between agents forms an interaction graph that may evolve as the system runs.

Nodes, Edges, and State

The three fundamental building blocks of any agent graph are nodes, edges, and state. Understanding each one deeply is essential before you start building complex systems.

Nodes are functions. In LangGraph, a node is a Python function that accepts the current graph state and returns a partial update to that state. The function signature looks like this:

```
def my_node(state: State) -> dict:
    # Read from state
    messages = state["messages"]
    # Perform computation
    result = some_processing(messages)
    # Return partial state update
    return {"result": result}
```

The key insight is that nodes do not return the full state. They return only the fields they want to change. The graph runtime merges these partial

updates into the current state using reducer functions. This design keeps nodes focused and composable – each node handles one responsibility and emits a clean delta.

Nodes can be synchronous or asynchronous. A sync node blocks until it completes; an async node yields control to the event loop while waiting for I/O, enabling concurrent execution of multiple async nodes within the same super-step. For LLM calls, database queries, and API requests, async nodes are almost always better because they allow the graph to overlap I/O-bound operations.

Nodes must be idempotent. Because graphs support checkpointing and recovery, a node may execute multiple times if the system crashes and resumes from an earlier checkpoint. An idempotent node produces the same result regardless of how many times it runs with the same input. This means nodes should not have side effects like sending emails or charging credit cards unless those side effects are themselves idempotent or are explicitly managed through compensation logic.

Edges define flow. Edges come in three flavors:

Static edges connect two nodes unconditionally. When node A completes, the graph always proceeds to node B. Static edges define the backbone of your workflow – the paths that always execute.

Conditional edges add routing logic. After a node completes, a routing function examines the state and decides which node to run next. This is how agents make decisions based on LLM outputs, tool results, or other computed values. The routing function receives the current state and returns the name of the next node (or a list of names for parallel execution).

Dynamic edges are created at runtime using the Command and Send APIs. A node can return a Command object that simultaneously updates state and directs execution to a specific next node. This is more flexible than conditional edges because the routing logic lives inside the node itself rather than in a separate routing function. The Send API creates parallel execution branches dynamically, spawning worker nodes with isolated state copies based on runtime data.

State is the bloodstream of the graph. State carries all information between nodes – conversation history, tool results, intermediate computations, error records, and accumulated knowledge. Without state, each node would

operate in isolation, and the graph would be nothing more than a collection of disconnected functions.

In LangGraph, state is defined by a schema that can be a TypedDict, a Pydantic model, or a dataclass. The schema declares the fields and their types, and optionally specifies reducer functions for each field. Reducers determine how updates from different nodes merge when they run concurrently.

The default behavior for any field without an explicit reducer is last-write-wins: if two parallel nodes both update the same field, the later write overwrites the earlier one. For most fields, this is fine – a status flag or a single result value does not need to accumulate. But for fields that collect data across multiple steps, you need reducers.

The most important built-in reducer is `add_messages`, which merges lists of chat messages by ID. When two nodes append messages to the conversation history, `add_messages` ensures that messages with the same ID are deduplicated (keeping the latest version) and new messages are appended. Without this reducer, parallel branches would overwrite each other's messages.

Custom reducers are equally important. If you have a field that accumulates findings from multiple research nodes, you would use `operator.add` to concatenate lists:

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

class ResearchState(TypedDict):
    messages: Annotated[list, add_messages]
    findings: Annotated[list[str], add]
    status: str
```

Here, `messages` uses the built-in deduplication reducer, `findings` concatenates strings from parallel nodes, and `status` uses last-write-wins (no reducer specified).

Acyclic vs Cyclic Execution Models

The presence or absence of cycles fundamentally changes what a graph can express and how it executes.

Acyclic graphs (DAGs) have a natural termination guarantee. Since there are no cycles, following edges from any node must eventually reach a terminal node. This makes DAGs easy to reason about: you know the workflow will finish, and you can compute the longest path to estimate execution time. DAGs are ideal for linear workflows with optional branching – data preprocessing pipelines, sequential approval chains, and simple if-then-else decision trees.

The limitation of DAGs is that they cannot express iteration. If an agent needs to retry a failed operation, refine an answer based on self-evaluation, or loop through a list of items, a DAG falls short. You would need to unroll the loop into a fixed number of steps, which wastes resources when fewer iterations suffice and fails when more are needed.

Cyclic graphs add loops, enabling iteration, retry, and refinement patterns. An agent that generates code, tests it, and retries on failure is a cycle: generate -> test -> (if failed) back to generate. A self-correcting agent that evaluates its own output and revises it forms a similar loop.

Cycles introduce two challenges that acyclic graphs avoid:

First, termination is not guaranteed. A cycle could theoretically run forever. Graph execution engines solve this with recursion limits – a maximum number of steps after which execution halts. LangGraph defaults to 1,000 steps, which is generous for most use cases but configurable based on your needs. Well-designed agents also include explicit termination conditions in their logic, such as "retry up to 3 times" or "stop when confidence exceeds 0.9."

Second, cycles complicate state reasoning. In a DAG, you can trace the value of any field through the graph's topology to understand its provenance. With cycles, a node may read values it wrote in a previous iteration, creating feedback loops that are harder to analyze statically. This is not a dealbreaker – many production systems use cyclic graphs successfully – but it means you need more discipline in state design and more thorough testing.

The execution model for cyclic graphs in LangGraph follows the Pregel-inspired super-step approach:

1. Identify all nodes that are eligible to run (their predecessor edges have been satisfied).
2. Execute all eligible nodes, potentially in parallel if they are async and have no mutual dependencies.
3. Merge all node outputs into the shared state using reducers.

4. Checkpoint the updated state.
5. Repeat from step 1 until no nodes are eligible or the recursion limit is reached.

This model treats each cycle iteration as a new super-step, with clean synchronization barriers between steps. It provides deterministic execution semantics even in the presence of cycles and parallel branches.

The Dataflow Paradigm

Dataflow programming is a paradigm where programs are represented as directed graphs, and execution is driven by data availability rather than sequential control flow. In a dataflow graph, each node (called an "actor" in the original literature) fires when all its input tokens are available. The output of one node becomes the input token for downstream nodes.

This model has several properties that make it attractive for agent systems:

Implicit parallelism. Since nodes fire as soon as their inputs are ready, independent branches of the graph execute concurrently without explicit threading or async code. The scheduler discovers parallelism automatically from the graph topology.

Explicit data dependencies. Every data dependency is visible as an edge in the graph. There are no hidden shared variables or global state – data flows along edges, and you can trace any value's origin by following edges backward.

Self-scheduling. The execution engine decides which node to run next based on data availability. There is no central program counter or fixed instruction sequence. This makes the model naturally distributed – different parts of the graph can execute on different machines as long as data tokens are communicated between them.

Separation of "what" from "how." The graph describes what computation to perform and how data flows between operations. The execution engine decides how to schedule, parallelize, and distribute that computation. This separation enables optimization without changing the program logic.

LangGraph's execution model shares spirit with dataflow but is not a pure dataflow system. In LangGraph, nodes execute based on control flow (edges indicate "run after") rather than purely on data availability. State is shared and

mutable (with reducer-mediated updates) rather than flowing as immutable tokens along edges. The hybrid approach gives you the structural clarity of graphs with the practical flexibility of imperative programming.

Understanding the dataflow paradigm helps you think about agent design in the right way. Instead of asking "what function do I call next?", you ask "what data does each step need, and which steps can run independently?" This shift in perspective leads to more parallelizable, more maintainable agent architectures.

Here is a minimal example that demonstrates the core concepts we have discussed. It creates a simple acyclic graph with two nodes and static edges:

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END


# Define the state schema
class SimpleState(TypedDict):
    input_text: str
    processed_text: str
    result: str


# Node 1: process the input
def process_node(state: SimpleState) -> dict:
    text = state["input_text"]
    return {"processed_text": text.upper()}


# Node 2: generate final result
def result_node(state: SimpleState) -> dict:
    processed = state["processed_text"]
    return {"result": f"Processed: {processed}"}


# Build the graph
builder = StateGraph(SimpleState)
builder.add_node("process", process_node)
builder.add_node("result", result_node)

# Connect nodes with static edges
builder.add_edge(START, "process")
builder.add_edge("process", "result")
builder.add_edge("result", END)

# Compile and run
graph = builder.compile()
```

```
output = graph.invoke({"input_text": "hello world"})
print(output["result"])  # Output: Processed: HELLO WORLD
```

This graph has three nodes (START, the two custom nodes, and END), connected by static edges. Execution flows linearly from START through process to result to END. The state carries data between nodes, and each node returns a partial update that the runtime merges into the shared state.

In the chapters ahead, you will see these same primitives – state, nodes, and edges – used to build increasingly sophisticated agent systems. The complexity comes not from new concepts but from composing these fundamentals in creative ways to express planning, memory, tool use, multi-agent collaboration, and everything else that defines modern AI agents.

Chapter 2: State Management in Agent Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Designing Agent State Schemas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

State Transitions and Immutability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Checkpointing and Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Persistent Memory Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

State Serialization and Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 3: Building Graph Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Custom Node Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Edge Routing and Conditionals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Graph Composition Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Error Handling at the Graph Level

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Building Reusable Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Framework Comparison: LangGraph, CrewAI, and AutoGen

Chapter 4: Tool Orchestration and Function Calling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Tool Definition and Registration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Tool Selection Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Multi-Step Tool Chains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Error Recovery and Retries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Structured Output Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 5: Reasoning Graphs and Chain-of-Thought Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chain of Thought as a Graph Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Tree of Thoughts and Branching Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Self-Correction Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Reflection and Critique Nodes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Iterative Refinement Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 6: Multi-Agent Interaction Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Agent Communication Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Supervisor and Hierarchical Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Debate and Consensus Mechanisms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Parallel Agent Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Emergent Multi-Agent Behaviors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 7: Workflow Orchestration at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Long-Running Agent Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Human-in-the-Loop Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Distributed Graph Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Workflow Composition and Subgraphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Production Deployment Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 8: Memory Systems for Agent Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Memory Architecture Taxonomy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Short-Term and Working Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Long-Term Persistent Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Episodic Memory and Retrieval

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Semantic Memory and Knowledge Extraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Memory Integration with Graph State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 9: RAG in Graph Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

RAG as a Graph Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Multi-Step Retrieval Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Query Decomposition and Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Synthesis and Aggregation Nodes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Evaluation and Quality Assurance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 10: Streaming, Async, and Real-Time Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Streaming Response Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Asynchronous Node Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Real-Time Agent Interaction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Concurrency Control in Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Latency Optimization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 11: Debugging, Observability, and Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Graph Execution Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Logging and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Unit Testing Graph Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Integration and End-to-End Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Debugging Tools and Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Automated Evaluation Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

CI/CD Pipeline Integration for Agent Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 12: Performance, Scalability, and Fault Tolerance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Performance Profiling and Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Horizontal Scaling Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Fault Tolerance and Circuit Breakers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Rate Limiting and Cost Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Reliability Engineering for Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 13: Security, Safety, and Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Threat Model for Agent Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Prompt Injection Defense Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Output Validation and Sanitization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Access Control and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Audit Trails and Compliance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Chapter 14: Advanced Patterns and Production Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

The Research Agent Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

The Customer Service Agent Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

The Code Generation Agent Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Hybrid Human-AI Workflow Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Building a Complete Production System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

Conclusion: The Future of Graph-Based AI Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/graphengineeringforaiagents.