LangChain vs LangGraph: Workflow and Agentic System Design

Introduction

LangChain is an open-source library designed to simplify the process of building applications powered by Large Language Models (LLMs). It provides modular and composable components that help developers build sophisticated LLM-based workflows efficiently.

LangChain at a Glance

- **Purpose:** To streamline the creation of LLM-based applications.
- **Core Idea:** Composable chains connecting models, prompts, retrievers, and memory.
- **Main Offering: Chains** – allowing sequential execution of multiple components.

Core Components of LangChain

1. **Model Components:** Provide a unified interface to interact with multiple LLM providers.
2. **Prompts:** Facilitate structured prompt engineering.
3. **Retrievers:** Fetch relevant documents or data from vector databases.
4. **Chains:** The key abstraction to build sequential workflows.

Applications of LangChain

- Chatbots and conversational systems.
- Text summarization or Q&A systems.
- Multi-step reasoning pipelines.
- Basic Retrieval-Augmented Generation (RAG) systems.

Workflow vs Agentic System

Key Distinction

Workflow: Predefined, developer-designed sequence of steps that an AI follows.
Agentic System: An autonomous system that dynamically decides which steps to execute based on the situation.

In a workflow, control is linear and scripted. In an agentic system, control is dynamic — the AI decides which tools to use, in what order, and when to stop.

Example: Automated Hiring Process

- In a workflow, developers design a static sequence: create JD → approve JD → post JD → collect applications.
- In an agentic system, an AI agent dynamically plans and executes each step, adjusting based on real-time data.

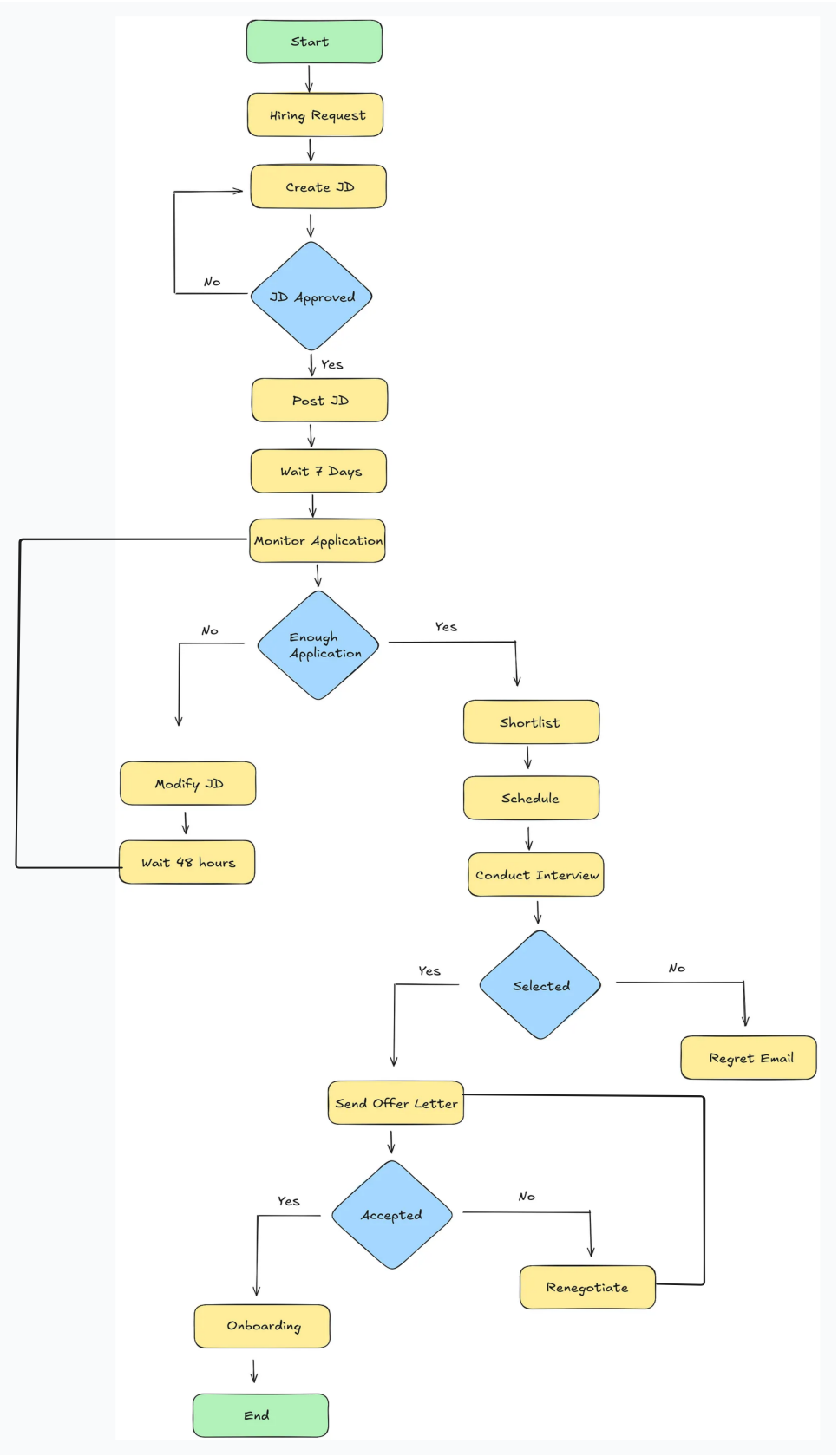

Figure 1: **Automated Hiring Process**

Challenges of LangChain in Complex Workflows

1. Control Flow Complexity

LangChain excels in linear workflows but struggles with:

- **Conditional Branches** (e.g., if JD approved → post, else revise).
- **Loops** (e.g., re-generate JD until approved).
- **Non-linear Jumps** (e.g., returning to earlier steps dynamically).

Limitation: LangChain requires extra "glue code" to manage such logic, increasing maintenance difficulty.

LangGraph's Solution

LangGraph models workflows as a **graph of nodes and edges**. Each node represents a task, and edges represent control flow. **Loops**, **conditional paths**, and **retries** are natively supported — no extra code required.

2. Handling State

Problem in LangChain: LangChain's "memory" is designed for conversational history, not structured workflow state. Developers must manually maintain dictionaries to track evolving variables (e.g., `jdApproved=true`).

LangGraph's Approach to State Management

LangGraph introduces a shared **state object**:

- Automatically passed between nodes.
- Mutable and globally accessible.
- Eliminates manual state tracking.

This enables clean and consistent handling of dynamic data throughout execution.

3. Event-Driven Execution

Sequential vs Event-Driven:

- **Sequential:** Runs continuously from start to finish.

- **Event-Driven:** Pauses at checkpoints, waits for external events (e.g., user input, time delay).

LangChain lacks built-in support for event-driven workflows — developers must manually manage pauses and resumptions.

LangGraph's Event-Driven Model

LangGraph supports pausing at any node and checkpointing the state. The workflow can later resume exactly where it left off after an external trigger (e.g., 7-day delay, candidate response).

4. Fault Tolerance

LangChain does not inherently handle failures or partial re-execution. If a step fails, the chain often restarts from the beginning.

LangGraph's Fault Recovery

- Built-in **retry mechanisms**.
- Automatic **checkpointing** after each node.
- Resume from point of failure without re-running previous steps.

Result: Improved reliability for long-running, real-world workflows.

5. Human-in-the-Loop Integration

Human-in-the-Loop (HITL) means pausing a workflow for human approval or feedback before proceeding.

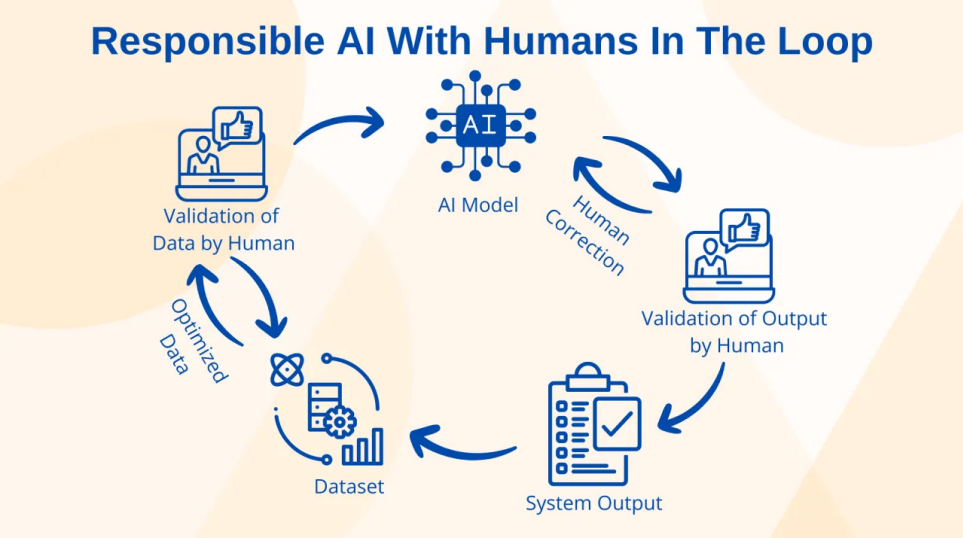

Figure 2: **HITL**

LangChain Limitation:

- No native pause/resume capability.
- Requires splitting chains and manual state passing.

LangGraph's HITL Support

LangGraph natively supports HITL:

- Save workflow state at checkpoints.
- Pause indefinitely for human decision.
- Resume seamlessly from the saved state.

6. Nested Workflows

Nested workflows occur when a workflow calls or triggers another workflow as a sub-process. For instance, during a hiring process, an "Interview Scheduling" sub-flow may include several internal steps — calendar coordination, candidate confirmation, and interviewer assignment — that function as an independent workflow.

LangChain Limitation:

- Lacks a structured way to compose or reuse sub-workflows.
- Developers must manually manage context passing between parent and child chains.
- Error handling or rollback between nested chains becomes cumbersome.

LangGraph's Nested Workflow Management

LangGraph supports the concept of **subgraphs**, enabling modular workflow composition:

- Each subgraph acts as an independent, reusable workflow.
- The parent graph can invoke, pause, or resume subgraphs dynamically.
- State is seamlessly shared across all levels of nesting.

Result: Scalable and maintainable system design where complex agentic behaviors emerge naturally from reusable sub-workflows.

7. Observability

Observability allows tracking, debugging, and auditing workflow behavior.

LangChain + LangSmith:

- Tracks LLM inputs/outputs, latency, token usage.
- *Limitation:* Does not monitor custom "glue code" logic.

LangGraph's Full Observability

When paired with LangSmith:

- Every node and transition is tracked.
- Full state history is observable.
- Provides complete auditability and debugging support.

Advantage: 100% visibility across the entire graph — no hidden logic.

What is LangGraph?

LangGraph is an orchestration framework that enables developers to build **stateful**, **multi-step**, and **event-driven** workflows using Large Language Models (LLMs). It is designed for both **single-agent** and **multi-agent** agentic AI applications.

Conceptual Overview

Think of **LangGraph** as a *flowchart engine for LLMs* — you define the steps (**nodes**), how they are connected (**edges**), and the logic that governs transitions. LangGraph automatically handles:

- **State management**
- **Conditional branching and looping**
- **Pausing and resuming execution**
- **Fault detection and recovery**

These capabilities make it ideal for building robust, production-grade agentic AI systems where dynamic decision-making and resilience are essential.

When to Use LangChain vs LangGraph

Use LangChain When:

- The workflow is **simple and linear**.
- You're building a **basic chatbot, summarizer, or RAG system**.
- State management and event-driven logic are minimal.

Use LangGraph When:

- Workflow includes **loops, branches, or conditions**.
- You need **Human-in-the-Loop** or **multi-agent coordination**.
- Execution is **asynchronous or event-driven**.
- Robust **state tracking, fault tolerance, and observability** are required.

Summary Table

Aspect	LangChain	LangGraph
Workflow Type	Linear / Sequential	Dynamic / Non-linear
State Handling	Manual	Built-in Stateful Object
Event-Driven Support	Limited	Native
Fault Tolerance	Minimal	Checkpointed Recovery
Human-in-the-Loop	Manual Integration	Native Pause & Resume
Observability	Partial (via LangSmith)	Full (via LangSmith + Graph)

Conclusion

LangChain remains the best choice for **simple, quick-to-build LLM pipelines**. However, as your application grows in complexity — involving multiple decisions, asynchronous tasks, or human inputs — **LangGraph** becomes the superior framework.

LangChain for simplicity, LangGraph for scalability.