Video 3 – Langchain Vs Langraph

Today's goals

1. Build intuition for **why LangGraph exists** (what problems it solves beyond LangChain).
2. Give a **technical overview** of LangGraph.
3. Compare **LangChain vs. LangGraph** and when to use each.

Quick LangChain recap

LangChain simplifies building LLM apps using modular blocks:

- **Models** (unified interface for OpenAI, Anthropic, Hugging Face, Ollama, etc.)
- **Prompts** (prompt templates & engineering)
- **Retrievers** (RAG: fetch relevant docs from vector stores)
- **Chains** (compose components into linear multi-step flows)
  With these, you can build chatbots, summarizers, multi-step pipelines, RAG apps, and simple tool-using "agents."

Workflow vs. Agent (important distinction)

- A **workflow** follows **predefined code paths** designed by the developer (static flowchart).
- An **agent** dynamically plans tools/steps and controls its own process (autonomous, changes run to run).

Using the automated hiring scenario (JD creation → approval → posting → waiting → monitoring → shortlist → schedule interviews → conduct → offer → renegotiate → onboarding), the speaker shows why this is **complex and non-linear**.

Why complex workflows are hard in LangChain (and how LangGraph helps)

1. **Control-flow complexity**
   - LangChain chains are mostly linear. Complex flows need **conditions, loops, and jumps**, which force you to write lots of custom "glue code."
   - **LangGraph** models workflows as a **graph** of **nodes** (tasks) and **edges** (transitions), with built-in branching, looping, and jumps—**no glue code**.

2. **State handling**
    - Complex workflows track evolving state (JD text, approval flags, counts, thresholds, candidates, offers, onboarding status). LangChain lacks a native key-value workflow state; you end up hacking dictionaries yourself.
    - **LangGraph is stateful**: each node receives and returns a shared, mutable **state** object (TypedDict/Pydantic). Nodes read/update it naturally.
3. **Event-driven execution (pause/resume)**
    - Real flows pause for time or external triggers (e.g., wait 7 days for applications, wait for candidate to accept). LangChain assumes synchronous, sequential runs. You'd split into multiple chains and pass state manually.
    - **LangGraph supports pausing** with **checkpointers** and **resuming** from saved state on external events.
4. **Fault tolerance**
    - Long-running flows need resilience to small faults (API hiccups) and big ones (server crash). LangChain offers no built-in recovery; you typically rerun from the start.
    - **LangGraph provides retries** for transient errors and **recovery from checkpoints**, resuming from the last completed node.
5. **Human-in-the-loop**
    - Many steps need human approval (e.g., approve JD). LangChain can take short synchronous input, but not indefinite waits.
    - **LangGraph treats human review as first-class**: pause indefinitely, store context, resume when approval arrives.
6. **Nested workflows (subgraphs)**
    - Complex nodes (e.g., "Conduct Interviews") can themselves be full workflows.
    - **LangGraph allows subgraphs**, enabling **multi-agent systems** and **reusable** mini-workflows (e.g., a generic "approval" subgraph used in many places). LangChain doesn't support this natively.
7. **Observability**

    - Observability (monitoring/debugging/auditing) is essential. **LangSmith** integrates well with LangChain, but won't see your custom glue code, so visibility is partial.
    - **LangGraph integrates tightly with LangSmith** to record node-level transitions, state diffs, messages, human-in-loop points—yielding **end-to-end traceability**.

What LangGraph is (succinct)

An **orchestration framework** for building **stateful, multi-step, event-driven** LLM workflows and both single-agent and multi-agent systems. Think of it as a **flowchart engine for LLMs** that handles state, branching/loops, pause-resume, and fault recovery.

When to use what

- **Use LangChain** for **simple, linear** flows: prompt chains, summarizers, basic RAG.
- **Use LangGraph** for **complex, non-linear** flows: conditions/loops, human-in-loop, multi-agent coordination, asynchronous/event-driven execution.

Do you still need LangChain?

Yes. **LangGraph is built on LangChain.** You still use LangChain components (models, prompts, retrievers, loaders, tools). LangGraph **orchestrates** them cleanly for complex production systems. They work **hand-in-hand**.

Bottom line: After this video, you should be able to look at a use case and decide whether a simple LangChain chain suffices or you need LangGraph's graph-based, stateful orchestration for a robust agentic workflow

Video 4 – Langraph Core Concepts

LangGraph LLM Workflows — Comprehensive Study Notes

A structured, at-a-glance reference for later stages of learning & implementation

---

1) What is LangGraph?

- **Definition:** An orchestration framework to **design and run intelligent, stateful, multi-step LLM workflows**.
- **Key idea:** Model workflows as a **graph** — *nodes* (tasks) and *edges* (control flow) — then **execute** the graph.
- **Why it matters:** Enables **parallelism, branching, loops, shared state, observability, and resumability** for production-grade agentic apps.

---

2) Foundations

2.1 Workflow vs. LLM Workflow

- **Workflow:** An ordered **series of tasks** to achieve a goal.
- **LLM Workflow:** A workflow where several tasks depend on LLMs (prompting, reasoning, tool calls, memory access, decision making).

2.2 Mental Model

- **Nodes = what to do** (a Python function per task)
- **Edges = when/where to go next** (sequential, conditional, parallel, loop)
- **State = the bloodstream** (shared, evolving data passed between nodes)

---

3) Common LLM Workflow Patterns

Pick the pattern that matches your problem. Mix & match when needed.

3.1 Prompt Chaining

- **What:** Call the LLM **multiple times in sequence**.

- **Use when:** Complex tasks that benefit from **decomposition** (e.g., *topic* → *outline* → *full report*).
- **Good practices:**
    - Insert **validators/guards** (e.g., word count, schema checks).
    - Save intermediate artifacts in **state** for debugging/reuse.

3.2 Routing

- **What:** A "router" LLM **classifies the task** and routes it to the right handler/model/tool.
- **Use when:** Queries span **different domains** (refunds vs. tech vs. sales).
- **Good practices:**
    - Keep **routing labels** explicit; log confidence.
    - Provide **fallback** or human handoff for low confidence.

3.3 Parallelization

- **What:** Split into **independent subtasks** and **run simultaneously**; then **aggregate**.
- **Use when:** Subtasks **don't depend** on each other (e.g., policy check, misinformation scan, sensitive-content check).
- **Good practices:**
    - Define a **merge policy** (e.g., weighted score, AND/OR thresholds).
    - Guard against **fan-out explosion** (limit parallel breadth).

3.4 Orchestrator–Workers

- **What:** Orchestrator **plans dynamically**; workers execute variable subtasks in parallel.
- **Use when:** The **plan depends on the query** (Scholar vs. News, API vs. DB, etc.).
- **Good practices:**
    - Capture the **plan** (tools, sources, criteria) into state for traceability.
    - **Timeouts and budgets** per worker to avoid stalls.

3.5 Evaluator–Optimizer (Iterative)

- **What: Generator** proposes, **Evaluator** accepts/rejects with feedback → **loop** until criteria met.
- **Use when:** Creative/open-ended tasks (emails, blogs, copywriting) that improve via **iteration**.
- **Good practices:**
  - Make **evaluation criteria explicit** (rubric, style, constraints).
  - Enforce **max iterations** / early-stop; persist all drafts in state.

---

4) Graphs, Nodes, and Edges

- **Nodes:** One task per node; implemented as **Python functions** that read & update **state**.
- **Edges:** Define control flow:
  - **Sequential:** A → B → C
  - **Conditional/Branching:** A → (B **if** cond else C)
  - **Parallel:** A → {B, C, D} (simultaneous)
  - **Loops:** A → ... → A (until stop condition)

Example Flow (Essay Practice — UPSC)

1. Generate topic → 2. User writes essay → 3. Evaluate on **clarity/depth/language** (0–5 each) → 4. Aggregate (max 15)
2. **Threshold** check (e.g., ≥10 pass) → 6a. Pass: congratulate → end
   6b. Fail: return **feedback** → optional **rewrite loop** → re-evaluate

---

5) State (Shared Memory)

- **Definition:** All workflow data **required for execution** and **evolving over time** (e.g., essay text, per-criterion scores, totals, decisions).
- **Behavior:**
  - **Shared:** Every node receives the **current state**.
  - **Mutable:** Nodes **update** parts of the state and pass onward.

- **Implementation:** Typically a **TypedDict** (or Pydantic model) defining allowed keys and types.
- **Tip:** Treat state keys as an **API**; version them if they will evolve.

---

6) Reducers (How Updates Apply)

- **Purpose:** Specify **per-key update policy** to avoid losing context/history.
- **Policies:**
    - **Replace:** Overwrite old value (good for latest numeric result or final label).
    - **Append/Add:** Preserve **history** (chat transcripts, multiple drafts).
    - **Merge/Custom:** Combine structured outputs (e.g., union lists, max/min, weighted aggregates).
- **When it matters:**
    - **Chatbots:** Keep all messages (append), not just the latest.
    - **Drafting loops:** Store every draft + feedback (append) to show **progress**.
    - **Parallel merges:** Define deterministic **merge** to reconcile partial results.

---

7) Execution Model (How it Runs)

- **Inspiration:** Google **Pregel** (graph processing).
- **Lifecycle:**
    1. **Define**: Nodes, edges, and **state schema**.
    2. **Compile**: Validate structure (no **orphan nodes**, consistent links).
    3. **Invoke**: Provide **initial state** to the **first node** → execution begins.
- **Message Passing:** Updated state moves along edges to activate next nodes.
- **Supersteps:** Execution proceeds in **rounds**; a superstep can include **multiple parallel node steps**.
- **Completion:** Stops when **no active nodes** remain and **no messages** are in flight.
- **Resumability:** Persist checkpoints; resume from last consistent point on failure.

8) Design & Implementation Checklist

- **Decompose the goal** into atomic tasks → map each task to a **node**.
- **Define edges**: sequential, conditional (branch), parallel, loop.
- **State schema**: name keys, types, defaults; decide **reducers** per key.
- **Validation & guards**: schemas, word/size limits, toxicity checks, cost/time budgets.
- **Observability**: log inputs/outputs, state diffs, routing decisions, tool calls.
- **Error handling**: retries, circuit breakers, fallbacks, human handoff.
- **Resilience**: idempotent nodes, checkpoints, resumability.
- **Performance**: batch where possible, cap parallelism, cache repeated calls.
- **Security**: sanitize inputs, restrict tool scopes, redact secrets in logs.

---

9) Pitfalls & Anti-Patterns

- **Losing context** by blindly replacing state values (use reducers wisely).
- **Unbounded parallelism** causing cost/time blow-ups (set limits/budgets).
- **Opaque routing** (log criteria/confidence; add fallbacks).
- **Infinite loops** in evaluators (max iterations, early-stops).
- **Orphan nodes** or dead edges (always compile/validate graph).

---

10) Quick Glossary

- **Node:** A task (Python function) that reads & writes state.
- **Edge:** A connection dictating next step(s) of execution.
- **State:** Shared, evolving data carried across nodes.
- **Reducer:** Per-key rule for applying updates to state.
- **Router:** An LLM (or rule) that decides which branch/handler to use.
- **Orchestrator:** The planner/assigner of dynamic subtasks; workers do the work.

- **Superstep:** A round that may include multiple parallel node executions.

---

11) Mini-Template (Planning Aid)

- **Goal:** …
- **Inputs:** …
- **Outputs / Success Criteria:** …
- **Nodes:**
    1. … (purpose, inputs, outputs)
    2. …
- **Edges:** A→B, B→{C,D} (cond), {C,D}→E (merge), E→(loop if not done)
- **State keys (+ reducer):**
    - messages (append), scores (merge), final_label (replace), …
- **Guards:** max tokens/time, schema checks, thresholds
- **Observability:** logs, metrics, traces
- **Failure/Recovery:** retries, backoff, resume from checkpoint

---

12) Worked Example (Essay Practice)

- **Nodes:** generate_topic → collect_essay → parallel score_clarity/score_depth/score_language → aggregate_score → branch (pass/feedback) → optional loop to collect_essay.
- **State keys:** topic (replace), essay_drafts (append), scores (merge), total_score (replace), feedback (append).
- **Reducers:** essay_drafts=append, scores=merge, feedback=append, total_score=replace.
- **Guards:** scoring bounds 0–5; threshold 10/15; max drafts/iterations.
- **Observability:** store each draft + score + feedback for learning curve.

---

13) Fast Evaluation Rubrics (for the Evaluator)

- **Clarity:** structure, coherence, thesis, flow (0–5)
- **Depth:** analysis, evidence, counterpoints (0–5)
- **Language:** grammar, vocabulary, tone (0–5)
- **Accept if:** total ≥ threshold; else provide targeted feedback bullets.

Keep this sheet nearby when designing new workflows. Start simple, make state/reducers explicit, log everything, and add loops/parallelism only where they pay off.