LangGraph (Conditional Workflows) — English Note-Summary with Key Topics

1) What this video covers

- **Context/recap:** You've already built:
  - **Sequential flows** (linear, one task after another).
  - **Parallel flows** (multiple independent tasks at the same time).
- **Today's focus: Conditional workflows**—flows that **branch based on a condition** (like `if/else`). Only **one branch** runs for a given input, then execution rejoins later.

---

2) Core ideas & terminology

2.1 Conditional workflow vs. Parallel workflow

- **Parallel:** split into multiple branches and run **all** those branches together; merge later.
- **Conditional:** split into branches but run **exactly one** branch, chosen by a **condition**; then continue.

2.2 How conditional routing works in LangGraph

- **State-first design:** define a `TypedDict` state of inputs/outputs your nodes will read/write.
- **Nodes are functions:** `state_in -> partial_state_out`.
- **Conditional edges:** instead of `add_edge`, use `add_conditional_edges(from_node, condition_fn)`. `condition_fn(state)` returns **the next node's name** (e.g., `"A"` or `"B"`), which determines which branch runs.

2.3 Why you'll use this often

- Real apps must **adapt** to data (e.g., input type, validation, sentiment, thresholds).
- Think of this as the **workflow equivalent** of `if/elif/else`.

---

3) Example A — Quadratic Equation Solver (Non-LLM, Conditional)

Objective

Given coefficients **a, b, c**, compute discriminant **D = b^2 − 4ac** and:

- if **D > 0** → two distinct real roots
- if **D = 0** → one repeated real root
- if **D < 0** → no real roots

State (TypedDict)

```
    a: float
    b: float
    c: float
    equation: str
    discriminant: float
    result: str           # message with roots / no real roots
```

Nodes (what each does)

- `show_equation` → build string like `"ax² + bx + c"`, store in `equation`.
- `calculate_discriminant` → compute `b*b - 4*a*c`, store in `discriminant`.
- `real_roots` → if D>0, compute `(-b ± √D) / (2a)`, write to `result`.
- `repeated_root` → if D=0, compute `-b / (2a)`, write to `result`.
- `no_real_roots` → set `result = "No real roots"`.

Routing function (decides the branch)

```
check_condition(state) -> "real_roots" | "repeated_root" | "no_real_roots"
# Uses the value of state["discriminant"]
```

Graph (ASCII)

```
START
  ▼
show_equation
  ▼
calculate_discriminant
  ├─ (D > 0) ─▶ real_roots ─┐
  ├─ (D = 0) ─▶ repeated_root ┤
  └─ (D < 0) ─▶ no_real_roots ┘
                              ▼
                             END
```

Takeaways

- **Conditional edges** appear as dotted arrows in the visualizer; **only one** will fire.
- Use **partial returns** (just the keys you change) to keep state merges clean.

---

4) Example B — Customer Review Handling (LLM, Conditional + Structured Output)

Goal

Given a **customer review**, reply differently based on **sentiment**:

- If **Positive** → generate a **warm thank-you** response.
- If **Negative** → first **diagnose** the issue (issue type, tone, urgency) via LLM **structured output**, then craft a **helpful resolution** message using those fields.

State (TypedDict)

```
review: str
sentiment: Literal["positive","negative"]
diagnosis: dict         # {issue_type, tone, urgency} for negative cases
response: str           # final reply text to customer
```

LLM setup (reliability matters)

- Use **structured outputs** via LangChain/Pydantic to guarantee exact fields:
  - `SentimentSchema`：`{ sentiment: Literal["positive","negative"] }`
  - `DiagnosisSchema`：
    `{ issue_type: Literal["ui","performance","bug","support","other"], tone: Literal["neutral","disappointed","frustrated","angry"], urgency: Literal["low","medium","high"] }`

Nodes

- `find_sentiment` (structured LLM) → sets `sentiment`.
- **Conditional routing** via `check_sentiment(state)`：
  - `"positive_response"` if positive
  - `"run_diagnosis"` if negative
- `positive_response` (plain LLM) → friendly thank-you; maybe asks for a website review.
- `run_diagnosis` (structured LLM) → fills `diagnosis` with `{issue_type, tone, urgency}`.
- `negative_response` (plain LLM) → crafts empathetic, specific reply using `diagnosis`.

Graph (ASCII)

```
START
  ▼
find_sentiment  --(sentiment == positive)-->  positive_response --► END
```

```
          \
           └--(sentiment == negative)--> run_diagnosis --> negative_response --> END
```

Prompting tips

- **Sentiment prompt:** "What is the sentiment of the following review (positive/negative)? …"
- **Diagnosis prompt (structured):** "Diagnose this negative review. Return fields: issue_type (UI/performance/bug/support/other), tone, urgency (low/medium/high). Review: …"
- **Response prompts (plain):**
  - *Positive:* "Write a warm thank-you reply to this review. Also invite feedback on our website."
  - *Negative:* "You are a support assistant. The user faced {issue_type}, sounded {tone}, urgency {urgency}. Write an empathetic, helpful resolution message with next steps."

---

5) Design patterns & checklists

5.1 Conditional pattern (reusable)

```
# Node functions: read from state, return partial dict def node(state: State) -> dict:
... def check_condition(state: State) -> str: # return the NEXT node's name based on state
return "branch_a" if ... else "branch_b" graph.add_node("start_node", node)
graph.add_conditional_edges("start_node", check_condition) # Then connect each branch to
the next common step or END
```

5.2 When to use structured outputs

- You need **exact fields** (e.g., `sentiment`, `score`, `issue_type`).
- You want to avoid brittle parsing (numbers as words, missing keys, etc.).

5.3 Debug checklist

- State lists **all** fields used/produced by nodes.
- Each node returns **only** the keys it modifies.
- Conditional function returns **valid node names**.
- Use **structured outputs** when LLM must return specific fields.
- Visualize after `compile()` to confirm edges (dotted = conditional).

---

6) Why conditional flows matter

- They let your agent **adapt** to inputs (like routing by type/quality/thresholds).
- They keep flows **clean and debuggable** compared to putting all logic inside one giant prompt.

- Combine with **parallel** and **sequential** blocks to model real production pipelines.

---

7) Quick reference (one-glance)

- **Sequential:** A → B → C
- **Parallel:** A → {B, C, D in parallel} → E
- **Conditional:** A → (if X) B → E
  A → (else) C → E