

LangGraph (Iterative / Looping Workflows) — English Note-Summary

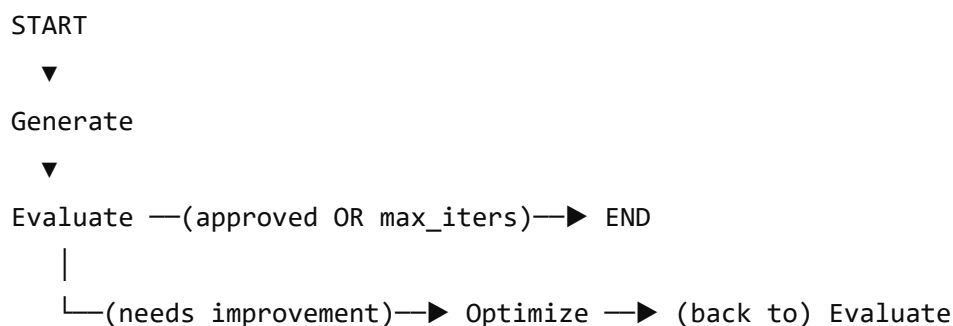
1) Quick recap & today's goal

- Already learned:
 1. Sequential flows ($A \rightarrow B \rightarrow C$)
 2. Parallel flows ($A \rightarrow \{B||C\} \rightarrow D$)
 3. Conditional flows ($A \rightarrow (\text{if } X) B \text{ else } C \rightarrow D$)
- Today: Iterative / looping workflows — run a **generate** → **evaluate** → **improve** cycle repeatedly until quality is good (or a stop condition is met).

2) What is an Iterative (Looping) Workflow?

- A workflow that **repeats a set of steps** to improve an artifact (text, plan, code, etc.).
- You define:
 - **Loop body**: the steps that repeat.
 - **Gate/condition**: when to **stop** or **continue** (approval, score threshold, or max iterations).

Visual shape (generic)



3) Real-world use case: Auto-posting with quality control

Problem: Creator posts on YouTube but lacks time to craft quality posts for X/LinkedIn/IG.

Risk: 1-shot LLM output may be mediocre or repetitive.

Solution: Build a loop that **generates**, **evaluates**, and, if needed, **optimizes** the post until it's good.

Targets & scope (example in video)

- Target platform: X (Twitter).
- Goal: produce a **short, original, funny** tweet on a given **topic**.

4) Architecture: 3 cooperating LLM roles

1. **Generator LLM** (e.g., creative, strong writing) → drafts the tweet.
2. **Evaluator LLM** (strict, rule-following) → judges against criteria, returns **structured** result:
 - `evaluation: Literal["approved", "needs_improvement"]`
 - `feedback: str`
3. **Optimizer LLM** (rewrite specialist) → revises the tweet based on evaluator **feedback**.

In a production build, you can mix providers/models specialized for each role; in the demo the same family is reused with different variants.

5) State design (TypedDict suggestion)

```
topic: str                # input topic for the tweet
tweet: str                # latest tweet draft
evaluation: Literal["approved", "needs_improvement"]
feedback: str             # latest evaluator feedback
iteration: int             # current iteration count
max_iteration: int        # safety stop (e.g., 5)

# Optional histories (use a reducer to append)
tweet_history: List[str]
feedback_history: List[str]
```

Use **partial state returns** from nodes to avoid merge conflicts.

6) Nodes & prompts (what each does)

A) generate_tweet

- **Prompt (system):** "You are a funny, clever Twitter influencer."
- **Prompt (human):** "Write a short, original, **hilarious** tweet on topic {topic} .
Avoid Q&A style, keep ≤280 chars, use observational humor/irony/sarcasm, simple daily English."
- **Returns:** { tweet } (+ append to tweet_history).

B) evaluate_tweet (uses structured output)

- **Prompt (system):** "You are a ruthless Twitter critic; evaluate by humor, originality, virality, format."
- **Prompt (human):**

- Hard **criteria** (reject if: Q&A style, >280 chars, cliché/traditional joke).
- Ask for **structured JSON** with:
 - `evaluation` : "approved" OR "needs_improvement"
 - `feedback` : strengths + weaknesses, actionable.
- **Returns**: { `evaluation`, `feedback` } (+ append `feedback_history`).

C) `optimize_tweet`

- **Prompt (system)**: "Punch up tweets for virality and humor using given feedback."
- **Prompt (human)**: "Improve this tweet using this feedback. Keep ≤ 280 , no Q&A."
- **Returns**: { `tweet`, `iteration` = `iteration` + 1 } (+ append to `tweet_history`).

7) Loop & routing logic

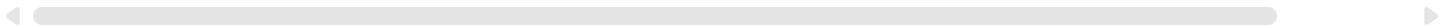
Graph (with loop)

START



```

Generate → Evaluate —(evaluation == "approved" OR iteration >= max)→ END
                |
                └(evaluation == "needs_improvement")→ Optimize → (back to)
  
```



Routing function (pseudo)

```

def route_evaluation(state): if state["evaluation"] == "approved" or state["iteration"]
>= state["max_iteration"]: return "END" else: return "optimize"
  
```

8) Structured outputs & reducers

- **Why structured outputs?** Reliability. You always get **exact fields** (`evaluation` , `feedback`) instead of brittle free-text.
- **Schemas (Pydantic)**:

```

class TweetEvaluation(BaseModel): evaluation: Literal["approved",
"needs_improvement"] feedback: str
  
```

- **Histories**: For `tweet_history` and `feedback_history` , use a **reducer** (e.g., `operator.add`) so parallel/iterative writes **append** rather than replace:
 - Return lists like { `tweet_history`: [tweet] } , which merge via list concatenation.

9) Guardrails & best practices

- **Max iterations** to avoid infinite loops (`max_iteration`).
 - Keep **prompts crisp & testable**; explicit reject rules boost evaluator consistency.
 - Return **partial state** from nodes (only the keys you change).
 - Log **history** (tweet + feedback) to audit improvement across iterations.
 - Use a more demanding Evaluator model (or stronger constraints) to force meaningful revisions.
 - In production: add **Human-in-the-Loop** before posting; integrate **platform APIs** for publishing.
-

10) One-page cheat-sheet (at a glance)

- **State:** `topic, tweet, evaluation, feedback, iteration, max_iteration, tweet_history[], feedback_history[]`
- **Nodes:** `generate → evaluate → (approved / needs_improvement)`
- **Loop:** if `needs_improvement` → `optimize` → `evaluate` (repeat)
- **Stop:** `approved` **or** `iteration >= max_iteration`
- **Tools:** Pydantic schemas for evaluator; reducer for histories; partial state updates.