AI Agent

LangGraph 워크플로우 제어

1. Loops, Branching, and Human-in-the-Loop

정적 vs 동적

LangChain:

- 정적(Static) 워크플로우
- 데이터가 흐르는 경로가 파이프라인처럼 미리 고정
- A -> B -> C로 이어지는 길은 한번 정해지면 바뀌지 않음
- 구조가 단순하고 예측 가능하지만 실행 중간에 동적으로 경로를 변경 불가

LangGraph:

- 동적(Dynamic) 워크플로우
- Agent가 현재의 State를 보고, 다음에 어떤 경로로 가야 할지를 실시간으로 결정
- A 다음에 B로 갈 수도, C로 갈 수도, 심지어 다시 A로 돌아올 수도 있음
- 훨씬 더 지능적이며, 현실 세계의 문제 해결 방식과 유사한 Agent 구성 가능

Agent의 판단

조건부 엣지 (Conditional Edges):

- LangGraph에서 동적 제어 흐름을 구현하는 핵심 기술
- Agent 스스로 작업 반복, 여러 대안 선택, 사람에게 도움 요청 등의 판단 기능

- 라우터 함수 (Router Function):
  - 현재 State를 입력받아, 다음에 이동할 노드를 반환하는 Python 함수

- graph.add_conditional_edges():
  - 라우터 함수와, 함수가 반환하는 각 문자열에 대응되는 실제 노드 매핑 메소드

Agent의 판단

```python
# 1. 라우터 함수 정의: State를 입력받아 다음 노드의 이름을 반환
def decide_next_step(state: AgentState) -> str:
    if state.get("error_count", 0) > 3:
        return "handle_error"  # 에러가 3번 초과하면 '에러 처리' 노드로
    elif state.get("is_complete", False):
        return "finish"      # 작업이 완료되었으면 '종료' 노드로
    else:
        return "continue_work" # 그렇지 않으면 '작업 계속' 노드로

# 2. 그래프에 조건부 엣지 추가
graph.add_conditional_edges(
    start_node_name="work_node", # 'work_node' 실행 후에 판단 시작
    condition=decide_next_step,  # 사용할 라우터 함수
    conditional_edge_mapping={   # 라우터 함수의 반환값과 실제 노드를 매핑
        "handle_error": "error_handling_node",
        "finish": END,
        "continue_work": "work_node" # 다시 work_node로 돌아가는 루프
      })
```

실무 구현 패턴

조건부 엣지로 만드는 3가지 핵심 제어 패턴

- 패턴 1: 루프 (Looping) -> Self-Correction & Iteration
- 패턴 2: 인간 개입 (Human-in-the-Loop) -> Safety & Approval
- 패턴 3: 에러 핸들링 (Error Handling) -> Robustness & Resilience

실무 구현 패턴

루프 (Self-Correction):

- 특정 조건이 충족될 때까지 Agent가 동일하거나 유사한 작업을 반복

- Self-Correction:
  - 초기 결과물이 목표에 미치지 못했을 때, 스스로 피드백하여 작업을 반복합니다.
  - (“보고서 초안의 논리가 부족하니, 다시 작성")

- Iterative Processing:
  - 여러 개의 하위 작업 목록을 하나씩 처리
  - ("질문 리스트가 빌 때까지, 하나씩 웹 검색을 수행")

- 라우터 함수가 특정 조건(if state['sub_questions'])을 만족 시
  - 현재 노드의 이름("research")을 다시 반환
  - 그래프의 흐름이 자기 자신에게 돌아오게 만듦

실무 구현 패턴

```python
# 라우터 함수
def should_continue_research(state: ResearchState) -> str:
    if state['sub_questions']: # State에 남은 질문이 있다면
        return "continue_research"
    else:
        return "synthesize"

# 조건부 엣지 설정
graph.add_conditional_edges(
    "research", # research 노드 실행 후 판단
    should_continue_research,
    {
        "continue_research": "research", # 'research'로 되돌아가 루프 형성
        "synthesize": "synthesis"})
```

실무 구현 패턴

인간 개입 (Human-in-the-Loop):

- Agent 워크플로우 중간에 실행을 일시정지(Interrupt)
- 사람의 검토 및 승인을 받은 후에만 다음 단계를 진행하도록 만드는 안전 장치

- 리스크 관리 측면에서
  - 데이터베이스 삭제, 고비용 API 호출 등의 작업 실행 전, 사람의 최종 확인으로 사고 방지

- 품질 보증 (QA) 측면에서
  - Agent가 생성한 계획이 올바른 방향인지 중간에 검토
  - 잘못된 방향으로 많은 시간과 비용을 낭비하는 것을 막음

- LangGraph는 이 기능들을 위해 Interrupt라는 체크포인트 클래스를 제공

실무 구현 패턴

```python
# 1. 그래프 컴파일 시 Interrupt 지점 설정
chain = graph.compile(
    checkpointer=memory,
    interrupt_before=["research"] # 'research' 노드 실행 직전에 멈춤
)

# 2. 첫 실행 및 일시정지
paused_state = chain.invoke(initial_state, config=config)
print(f"Agent의 계획: {paused_state['sub_questions']}")

# 3. 사용자 입력 받기
user_approval = input("이 계획을 승인하시겠습니까? (y/n): ")

# 4. 작업 재개
if user_approval == 'y':
    # invoke(None, ...)을 호출하여 멈췄던 지점부터 다시 실행
    final_state = chain.invoke(None, config=config)
```

실무 구현 패턴

에러 핸들링:

- 네트워크 불안정, 외부 API 서버 다운 등의 시나리오
- Tool 호출 실패, API 에러 등 문제 발생 시, 스스로 복구를 시도하거나 대체 경로를 선택
- Robust Agent를 위해서는 실패 상황을 미리 예측하고 대응하는 에러 핸들링 로직이 필수적

- State와 조건부 엣지를 조합하여 구현
- State에 retry_count와 같은 에러 추적용 변수를 추가
- 라우터 함수가 이 변수 값을 확인하여 다음 행동을 결정

실무 구현 패턴

```python
# State에 'retries' 추가
class RobustAgentState(TypedDict):
    # ... other fields
    retries: int

# 실패를 시뮬레이션하는 노드
def research_step(state: RobustAgentState):
    try:
        # ... Tool 호출 ...
        return {"retries": 0} # 성공 시 초기화
    except Exception as e:
        return {"retries": state.get("retries", 0) + 1} # 실패 시 1 증가

# 재시도 횟수를 확인하는 라우터
def router(state: RobustAgentState):
    if state.get("retries", 0) > 3:
        return "give_up"
    # ... 다른 조건들 ...

# 그래프에 'give_up' 경로 추가
graph.add_conditional_edges(..., {"give_up": END})
```