생성형 AI 기반 RAG 개발자 과정

LangGraph
상태 기반 워크플로우

Session 27 / 33 — Day 4

StateGraph · checkpoint · conditional edges · self-correction · monitoring

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

출처: Day 4 LangGraph

이번 시간을 마치면 — 원리

원리

- ✅ **StateGraph** = 상태 머신 모델 — 노드·엣지·상태
- ✅ **TypedDict / Pydantic** 으로 state schema 정의
- ✅ **조건부 엣지** — LLM 결정 / 함수 결정으로 분기
- ✅ **Checkpoint** — 메모리 영속화 + 시간여행 디버깅
- ✅ **Self-correction loop** — 평가 → 재시도 패턴
- ✅ **HITL (Human-in-the-loop)** — 그래프 중간 사용자 승인

이번 시간을 마치면 — 실습

실습

- ✅ 기본 그래프 한 노드 (`1.basic_graph.py`)
- ✅ 메모리 체크포인팅 (`2.memory_checkpointing.py`)
- ✅ 조건부 분기 — Router 패턴 (`3.conditional_branching.py`)
- ✅ Tool cyclic graph (`4.tools_cyclic_graph.py`)
- ✅ 자가교정 루프 (`6.self_correction_loop.py`)
- ✅ AgentOps 모니터링 — LangSmith 연동 (`7.agentops_monitoring.py`)

> 📂 소스: `2.langchain/9.langgraph/`

PART A

PART A
LangGraph — 상태 머신 모델

그래프로 LLM 워크플로우 그리기 — 약 20분

`create_agent` 의 한계와 LangGraph 의 역할

`create_agent` 의 강점

- `create_agent(llm, tools)` 한 줄로 ReAct 루프 자동 구성
- `{"messages": [...]}` 형식 입력 → 도구 호출·결과·답변이 messages 에 누적
- `system_prompt` 로 도구 사용 지침·성격 지정
- 메시지 히스토리 관리 (Reason → Act → Observe → 반복)

`create_agent` 로 구현이 어려운 것

- "두 번째 도구 호출 전에 사람 승인" — Human-in-the-loop
- "결과를 평가해서 점수 7점 미만이면 재시도" — Self-correction
- "병렬로 3 검색 → 가장 좋은 1개 선택" — Parallel + ranking
- "도구 호출 도중 외부 이벤트로 일시정지" — Pause/resume
- "전체 그래프 어디서든 재시작" — Checkpoint/replay

LangGraph 의 모델 — State · Node · Edge

3가지 핵심 추상화

추상화	역할
State (상태)	그래프 전반에 흐르는 데이터 (TypedDict 또는 Pydantic)
Node (노드)	state 를 입력받아 state 일부를 갱신하는 함수
Edge (엣지)	노드 간 연결 — 정적 또는 조건부

구조도

```
State 초기값
   ↓
[Node A] → state 갱신
   ↓ Edge (정적)
[Node B] → state 갱신
   ↓ Edge (조건부 — LLM 결정)
   ├→ [Node C1]
   └→ [Node C2]
        ↓
       END
```

한 노드 그래프

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END, MessagesState
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# ═══ 1. 상태 schema 정의 ===
# MessagesState 는 미리 정의된 typeddict {"messages": List[BaseMessage]}
graph = StateGraph(state_schema=MessagesState)

# ═══ 2. 노드 함수 ===
def call_model(state: MessagesState):
    messages = state["messages"]
    sys_msg = SystemMessage("당신은 친절한 AI 비서입니다.")
    response = llm.invoke([sys_msg] + messages)
    return {"messages": response}    # state 의 messages 키에 append 됨

# === 3. 그래프 구성 ===
graph.add_node("model", call_model)
graph.add_edge(START, "model")
graph.add_edge("model", END)

# ═══ 4. 컴파일 ===
app = graph.compile()

# ═══ 5. 실행 ===
result = app.invoke({"messages": [HumanMessage("LangGraph 이 뭐예요?")]})
```

출처: Day 4 LangGraph

한 노드 그래프 — State 갱신 규칙

State 갱신 규칙

- 노드는 state 전체가 아닌 **부분 dict 반환**
- `messages` 같은 list 필드는 reducer 에 의해 **append**
- 새 키는 추가, 기존 키는 갱신

> 💡 `MessagesState` 의 `messages` 가 자동으로 list-append 되는 동작은 LangGraph 의 reducer (`add_messages`) 에 기인한다.

TypedDict — 유연 · Pydantic — 엄격

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages

# ═══ TypedDict (간단) ═══
class MyState(TypedDict):
    question:  str
    documents: list[dict]       # 검색 결과
    answer:    str
    quality:   float            # 0~1 평가 점수
    attempts:  int

# ═══ Annotated reducer (list append 등 자동) ═══
class MessagesPlus(TypedDict):
    messages: Annotated[list, add_messages]   # 자동 append
    user_id:  str

# ═══ Pydantic (엄격) ═══
from pydantic import BaseModel, Field

class StrictState(BaseModel):
    question:  str = Field(min_length=1)
    documents: list[dict] = Field(default_factory=list)
    answer:    str | None = None
    quality:   float = Field(ge=0, le=1, default=0)
    attempts:  int = Field(ge=0, le=5, default=0)

graph = StateGraph(state_schema=StrictState)
# Pydantic 은 노드 진입 시마다 검증 → 안전하나 약간 느림
```

State 가 노드를 거치며 채워지는 모습

State 가 흐르는 모습

```
초기 invoke:    {"question": "X?", "attempts": 0}
node "retrieve" 후: {"question": "X?", "documents": [{...}, ...], "attempts": 0}
node "generate" 후: {"question": "X?", "documents": [...], "answer": "...", "attempts": 0}
node "evaluate" 후: {"question": "X?", "documents": [...], "answer": "...", "quality": 0.85, "attempts": 1}
```

PART B

PART B
조건부 엣지 — 분기와 루프

> if/else 와 while 을 그래프로 — 약 15분

3.conditional_branching.py — State · 노드

```
from typing import Literal
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    question: str
    category: str       # "math" | "search" | "chat"
    answer:   str

def classify(state: State) → dict:
    """LLM 으로 질문을 분류."""
    r = llm.invoke(f"""다음 질문의 카테고리:
질문: {state['question']}
"math" / "search" / "chat" 중 하나만 출력""").content.strip()
    return {"category": r}

def handle_math(state: State) → dict:
    """수학 질문 처리."""
    return {"answer": f"계산: {eval(state['question'])}"}

def handle_search(state: State) → dict:
    """검색 질문 처리."""
    return {"answer": f"검색 결과: ..."}

def handle_chat(state: State) → dict:
    """일반 대화."""
    return {"answer": llm.invoke(state["question"]).content}
```

`3.conditional_branching.py` — 라우터 함수

```
# === 라우터 함수 ===
def route_by_category(state: State) → Literal["math", "search", "chat"]:
    return state["category"]
```

라우터 함수의 시그니처

- 입력: 현재 state
- 출력: 다음 노드 이름 (str) — 매핑 dict 의 키

> 💡 라우터는 state 를 받아 **다음 노드 이름**만 반환하는 순수 함수다. 그래프 구성은 다음 슬라이드에서 다룬다.

3.conditional_branching.py — 조건부 엣지 연결

```
# === 그래프 구성 ===
g = StateGraph(State)
g.add_node("classify", classify)
g.add_node("math",     handle_math)
g.add_node("search",   handle_search)
g.add_node("chat",     handle_chat)

g.add_edge(START, "classify")
g.add_conditional_edges(
    "classify",                    # ← 어떤 노드에서
    route_by_category,             # ← 라우터 함수
    {                              # ← 매핑 (반환값 → 다음 노드)
        "math":   "math",
        "search": "search",
        "chat":   "chat",
    },
)
g.add_edge("math",   END)
g.add_edge("search", END)
g.add_edge("chat",   END)

app = g.compile()
print(app.invoke({"question": "2 + 2"})["answer"])
```

조건부 엣지를 이루는 3요소

조건부 엣지의 3요소

- **출발 노드** — `"classify"` 에서 분기 시작
- **라우터 함수** — `route_by_category` 가 다음 노드 결정
- **매핑 dict** — 반환값을 실제 노드 이름으로 변환

> 💡 매핑 dict 를 생략하면 라우터 함수의 반환값이 곧 노드 이름이 된다.

`4.tools_cyclic_graph.py` — ReAct 루프 직접 구현

```
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage
from langgraph.graph import MessagesState, StateGraph, START, END
from langgraph.prebuilt import ToolNode

# ═══ 도구 ===
@tool
def get_weather(city: str) → str:
    return f"{city}: 20°C, 맑음"

tools = [get_weather]
llm_with_tools = ChatOpenAI(model="gpt-4o-mini").bind_tools(tools)

# ═══ 노드 ===
def call_llm(state: MessagesState):
    return {"messages": llm_with_tools.invoke(state["messages"])}

tool_node = ToolNode(tools)   # tool_calls 를 자동 실행해 ToolMessage 반환
```

> ⬇️ 코드가 길어 다음 장에 이어집니다.

`4.tools_cyclic_graph.py` — ReAct 루프 직접 구현 (이어서)

```
# === 라우터 — tool_calls 가 있으면 도구 실행, 없으면 END ===
def should_continue(state: MessagesState) → Literal["tools", END]:
    last = state["messages"][-1]
    if last.tool_calls:
        return "tools"
    return END

# === 그래프 ===
g = StateGraph(MessagesState)
g.add_node("llm",   call_llm)
g.add_node("tools", tool_node)

g.add_edge(START, "llm")
g.add_conditional_edges("llm", should_continue, ["tools", END])
g.add_edge("tools", "llm")    # ← 루프! 도구 후 다시 llm

app = g.compile()
```

사이클이 구성하는 패턴

- `llm` 호출 → tool_calls 존재 → `tools` 실행 → 다시 `llm` → tool_calls 없음 → END
- 이 구조가 곧 **ReAct 의 LangGraph 구현체** (`create_agent` 내부)

> 🎯 이 사이클이 모든 Agent 작동 원리의 핵심이다.

PART C

PART C
Checkpoint — 그래프의 영속화

> 시간여행 디버깅과 멀티턴 대화 — 약 15분

2.memory_checkpointing.py 패턴

```
from langgraph.checkpoint.memory import MemorySaver
# 또는 langgraph.checkpoint.sqlite import SqliteSaver (영속)
# 또는 langgraph.checkpoint.postgres import PostgresSaver (운영)

# 직접 만든 graph 든 create_agent 든 — checkpointer 한 줄로 메모리 활성화
memory = MemorySaver()
app = graph.compile(checkpointer=memory)
# agent = create_agent(llm, tools, checkpointer=memory)  ← 에이전트도 동일

# === 같은 thread_id 로 호출 → state 가 누적 ===
config = {"configurable": {"thread_id": "alice"}}

# 첫 호출
app.invoke({"messages": [HumanMessage("내 이름은 앨리스야")]}, config=config)

# 두 번째 호출 — 같은 thread_id → 이전 state 자동 복원
result = app.invoke({"messages": [HumanMessage("내 이름이 뭐였지?")]}, config=config)
print(result["messages"][-1].content)
# → "앨리스 님이라고 하셨습니다"
```

> 💡 **기억 + 도구 사용 결합** — "내가 사는 곳 날씨?" 같은 질의에서 앞선 발화에 기억된 "서울" 을 도구 인자로 자동 보완한다. 멀티턴 에이전트의 핵심 동작이다.

Checkpointer 구현체와 세션 키

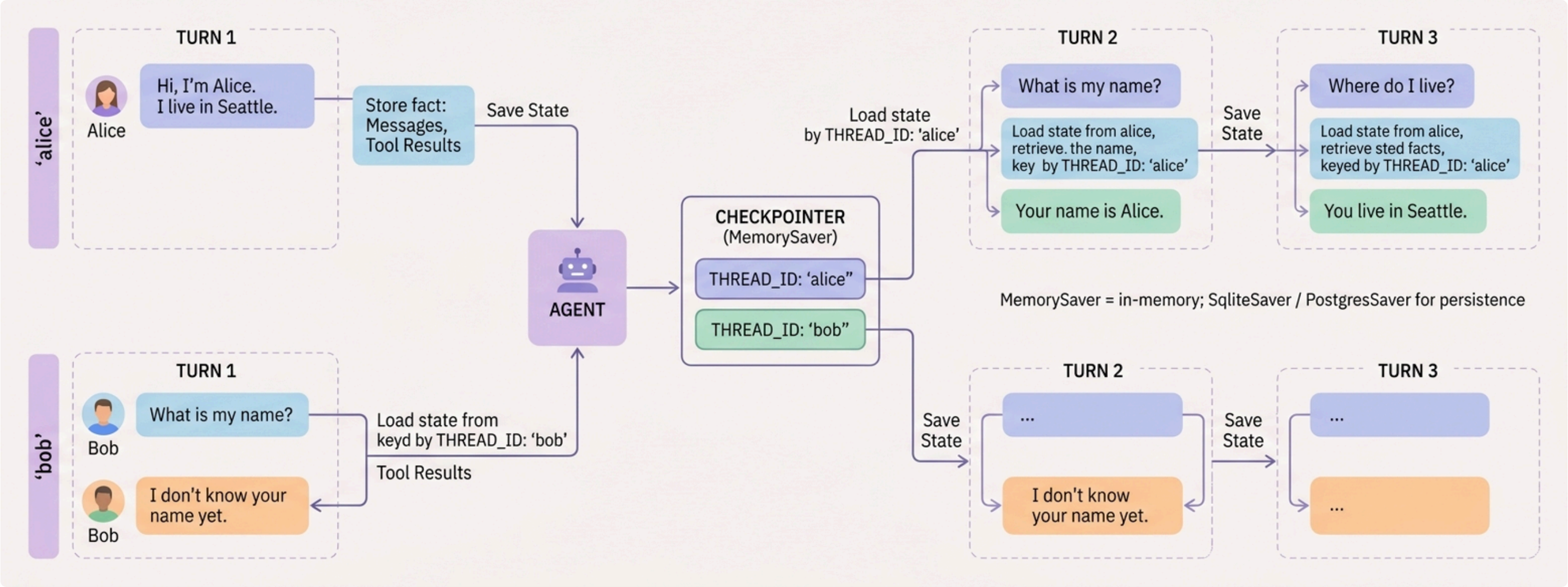
Checkpointer 종류

구현	영속성	용도
`MemorySaver`	프로세스 메모리	개발·테스트
`SqliteSaver`	로컬 파일	단일 서버 운영
`PostgresSaver`	DB	멀티 서버·고가용성
`RedisSaver` (3rd party)	Redis	고성능

thread_id 의 의미

- **세션 구분 키** — 대화형 RAG 의 `session_id` 와 동일 컨셉
- **같은 thread_id** = 맥락 유지 — 이전 대화·도구 결과를 이어받음
- **다른 thread_id** = 완전 격리 — alice 의 정보를 bob 은 전혀 모름
- 웹 서비스는 사용자/세션마다 thread_id 부여

thread_id 가 만드는 대화 메모리 격리



thread_id별 대화 메모리 격리 · 같은 thread는 이전 대화·도구 결과를 이어받고 다른 thread는 독립 세션 · MemorySaver 체크포인트로 상태 보존

Checkpoint 기반 디버깅 — 시간여행

```
# === 1. 전체 history 조회 ===
history = list(app.get_state_history(config))
for snapshot in history:
    print(f"step={snapshot.metadata.get('step', '?')}")
    print(f"  state.messages count: {len(snapshot.values.get('messages', []))}")
    print(f"  next: {snapshot.next}")    # 다음 실행 노드

# === 2. 특정 step 으로 되돌리기 ===
target_snapshot = history[3]
new_config = target_snapshot.config

# 해당 시점에서 재시작 (다른 입력으로)
app.invoke({"messages": [HumanMessage("다른 질문")]}, config=new_config)

# ═══ 3. 특정 노드 결과 수정 후 재실행 ===
# 운영 사고 대응에 강력함
app.update_state(
    config=target_snapshot.config,
    values={"answer": "수정된 답변"},   # state 일부 덮어쓰기
```

활용 시나리오 3가지

1. **디버깅** — 사용자 클레임 "이상한 답을 줬어요" → trace 따라가서 어느 노드에서 잘못됐는지
2. **A/B 테스트** — 같은 입력을 어느 분기점부터 다른 경로로
3. **사용자 편집** — "이전 답변에서 X 부분을 수정하고 싶어요" → state 갱신 후 재시작

PART D

PART D
Self-correction — 평가 → 재시도

> 품질 보장 패턴 — 약 15분

6.self_correction_loop.py — 생성 · 비평 노드

```
class CritiqueState(TypedDict):
    question:  str
    answer:    str
    critique:  str
    score:     float
    attempts:  int

MAX_ATTEMPTS = 3
THRESHOLD = 0.8

def generate(state: CritiqueState) → dict:
    """답변 생성 (또는 재생성 — critique 반영)."""
    prompt = f"""질문: {state['question']}
이전 답변: {state.get('answer', '없음')}
이전 비평: {state.get('critique', '없음')}
```

⬇️ 코드가 길어 다음 장에 이어집니다.

6.self_correction_loop.py — 생성 · 비평 노드 (이어서)

```
위 비평을 반영해 더 나은 답변:"""
    return {
        "answer":   llm.invoke(prompt).content,
        "attempts": state.get("attempts", 0) + 1,
    }

def evaluate(state: CritiqueState) → dict:
    """답변 자동 비평 — 점수 + 비평 1문장."""
    prompt = f"""다음 답변을 0~1 점수로 평가 + 비평 1문장:
질문: {state['question']}
답변: {state['answer']}

출력 형식:
score: 0.X
critique: ..."""
    text = llm.invoke(prompt).content
    score = float(re.search(r"score: ([\d.]+)", text).group(1))
    critique = re.search(r"critique: (.+)", text).group(1)
    return {"score": score, "critique": critique}
```

💡 `generate` 는 이전 `critique` 를 프롬프트에 넣어 **재생성**한다. 루프를 돌수록 답변이 비평을 반영해 개선된다. 분기·실행은 다음 슬라이드.

6.self_correction_loop.py — 재시도 분기 · 그래프

```
def should_retry(state: CritiqueState) → Literal["generate", END]:
    if state["score"] ⩾ THRESHOLD:
        return END              # 품질 충족 — 종료
    if state["attempts"] ⩾ MAX_ATTEMPTS:
        return END              # 한도 도달 — 그냥 종료
    return "generate"           # 점수 미달 — 재생성

# === 그래프 ===
g = StateGraph(CritiqueState)
g.add_node("generate", generate)
g.add_node("evaluate", evaluate)

g.add_edge(START, "generate")
g.add_edge("generate", "evaluate")
g.add_conditional_edges("evaluate", should_retry, ["generate", END])

app = g.compile()

result = app.invoke({"question": "RAG 의 핵심을 한 단락으로", "attempts": 0})
print(f"최종 점수: {result['score']}")
print(f"시도: {result['attempts']}회")
```

효과 (대략적 수치)

- 토큰 1.5~3배 사용 ↔ 답변 품질 평균 15% 향상
- 적용처: 코드 생성, 보고서, 번역 — 단번에 높은 품질이 요구되는 작업

그래프 중간에 사람 승인

```
from langgraph.types import interrupt

def review_before_send(state: State) → dict:
    """이메일 발송 전 사람 승인 — 사람이 입력할 때까지 일시정지."""
    user_decision = interrupt({
        "draft": state["draft"],
        "to": state["recipient"],
        "prompt": "발송하시겠습니까? approve/edit/reject 중 답변.",
    })
    return {"human_decision": user_decision}

# ═══ 컴파일 시 interrupt_before / interrupt_after ═══
app = graph.compile(
    checkpointer=memory,
    interrupt_before=["send_email"],   # 이 노드 직전에 일시정지
)

# 그래프 실행 → interrupt 시 멈춤
result = app.invoke(initial_input, config=config)
# result.next == ("send_email",) — 다음에 실행할 노드 정보

# 사람이 확인 후 재시작
app.invoke(None, config=config)   # interrupt 이후부터 계속
# 또는 state 수정 후 재시작
app.update_state(config, {"draft": "수정된 본문"})
app.invoke(None, config=config)
```

HITL 적용 시나리오

HITL 이 효과적인 상황

- **외부 작업 트리거** — 이메일·결제·코드 배포 직전
- **위험 도구 호출** — `rm -rf`, DB DROP, 외부 API 호출
- **민감 정보 요청** — 의료·법률 답변 발송 전
- **개인화 학습** — 사용자 피드백을 다음 결정에

PART E

PART E
AgentOps — 관측성

> LangSmith 로 trace 자동 수집 — 약 12분

Agent 동작 검증의 과제

Agent 운영의 어려움

- LLM 출력은 **비결정적** — 같은 입력에 다른 결과
- 도구 호출이 N단계 — 어디서 잘못됐는지 추적 어려움
- 비용·지연시간이 호출마다 다름
- 사용자가 "답이 이상해요" 라고만 신고 → 재현 불가능

필요한 관측성 4가지

측면	측정
Trace	모든 LLM·도구 호출의 입출력
Latency	각 노드/도구 응답 시간
Cost	호출별 토큰·비용 누적
Quality	사용자 피드백 (좋아요/싫어요) + 자동 평가

관측성 도구 비교

주요 도구

도구	강점	가격
LangSmith	LangChain 통합 ★★★★★	무료 5K trace/월
LangFuse	오픈소스, 셀프호스팅	무료
Helicone	OpenAI 프록시 형태	무료 10K/월
Arize Phoenix	오픈소스, RAG 평가 강	무료
Weights & Biases	ML 분야 강자, 통합 좋음	무료/유료

> 본 코스 권장: **LangSmith** (LangChain 생태계 가장 깔끔)

7.agentops_monitoring.py 패턴

```
# .env 에 추가
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=ls__...                    # smith.langchain.com 에서 발급
LANGCHAIN_PROJECT=genai-course-day4          # 프로젝트 이름 (자동 생성)
```

```
from dotenv import load_dotenv
load_dotenv()    # 위 4 변수 자동 적용

# 이후 LangChain 호출이 모두 자동 trace
# → smith.langchain.com 에서 실시간 확인
result = agent.invoke({"messages": [HumanMessage("RAG 설명")]})

# ═══ 수동 trace 추가 ===
from langsmith import traceable

@traceable(name="custom_processing")
def process_query(q: str) → str:
    # 일반 Python 코드도 trace
    return q.upper()
```

LangSmith 화면에서 보이는 것

LangSmith 화면에서 보이는 것

- **Run tree** — Agent invoke → llm → tools → llm 의 전체 트리
- **Latency** — 각 단계 ms 단위
- **Tokens / Cost** — 호출별 비용 + 누적
- **Input / Output** — 모든 메시지 본문
- **Errors** — 실패 자동 강조
- **Datasets** — 좋은 결과를 평가 데이터로 모으기
- **Evaluations** — 자동 평가 metric (RAGAS 통합)

Agent 답변에 좋아요/싫어요 받기

```
from langsmith import Client

client = Client()

def chat_with_feedback(question: str) → dict:
    """Agent 호출 + run_id 함께 반환."""
    result, run_id = agent.invoke_with_run_id({...})
    return {"answer": result["messages"][-1].content, "run_id": run_id}

# ═══ 사용자 응답을 feedback 으로 저장 ===
def add_feedback(run_id: str, score: int, comment: str = ""):
    """score: 1 = 좋음, -1 = 나쁨."""
    client.create_feedback(
        run_id=run_id,
        key="user_thumbs",
        score=score,
        comment=comment,
    )

# 사용 예 — Flask 엔드포인트
@app.route("/api/feedback", methods=["POST"])
def feedback():
    data = request.json
    add_feedback(data["run_id"], data["score"], data.get("comment", ""))
    return {"ok": True}
```

피드백이 만드는 운영 사이클

피드백이 만드는 사이클

```
배포 → Agent 호출 → 사용자 피드백 (👍/👎)
                          ↓
              LangSmith 대시보드에 누적
                          ↓
               낮은 점수만 따로 모아 분석
                          ↓
              프롬프트·도구·평가 메트릭 개선
                          ↓
                    배포 (반복)
```

> 🎯 **이 사이클이 운영 Agent 의 핵심**이다. 단일 배포로 종료되지 않고 피드백 → 개선을 반복한다.

PART F

PART F
미니 실습

> 자가교정 또는 HITL 그래프 — 약 30분

실습 과제 (30분) — 선택지

선택 1 — 자가교정 그래프 (코드 리뷰 봇)

- 입력: Python 함수 코드
- 노드: `review` (LLM 코드 분석) → `score` (점수 0~10) → 7 미만이면 → `improve` → 다시 `review`
- 최대 3회 반복

선택 2 — HITL 그래프 (이메일 어시스턴트)

- 입력: 이메일 작성 요청
- 노드: `draft` → `review_before_send` (사람 승인 대기) → `send`
- 사용자가 reject/edit 시 다시 draft 로

선택 3 — 내 도메인 그래프

- Session 9 의 워크플로 자동화를 LangGraph 로 재작성
- 조건부 분기·체크포인팅·HITL 1개 이상 포함

실습 과제 (30분) — 구현 단계

구현 단계

1. **State schema** 설계 (5분) — 어떤 데이터가 노드 간 흐르나
2. **노드 함수들** (10분)
3. **그래프 구성** + 조건부 엣지 (10분)
4. **Checkpoint + 실행** — 같은 thread_id 로 2번 호출해 메모리 확인 (5분)

결과 보고

- 그래프 다이어그램 (텍스트 형식 허용)
- 실행 trace 1건 (LangSmith 스크린샷 또는 print 출력)
- 가장 어려웠던 부분 한 줄

핵심 9가지

1. ✅ LangGraph = **State + Node + Edge** 의 상태 머신
2. ✅ TypedDict (간단) vs Pydantic (엄격) 으로 state schema
3. ✅ 조건부 엣지 — Router 패턴, 루프 (ReAct 직접)
4. ✅ `MessagesState` + `add_messages` reducer — list 자동 append
5. ✅ **Checkpoint** = `checkpointer` 한 줄로 메모리 활성화 + thread_id 별 격리
6. ✅ 시간여행 디버깅 — `get_state_history`, `update_state`
7. ✅ **Self-correction loop** — 평가 → 점수 → 재시도
8. ✅ **HITL** — `interrupt` 로 사람 승인 대기
9. ✅ **LangSmith** — 한 줄 환경변수로 trace 자동 수집

실습 과제 — 직접 해보기

필수

- [ ] LangGraph 1개 작성 — 조건부 분기 또는 자가교정 루프 포함
- [ ] LangSmith 가입 + 위 그래프의 trace 1건 캡처

옵션

- [ ] **HITL** 노드 추가 — 사용자 입력 받는 그래프
- [ ] **PostgresSaver** 로 멀티 서버 checkpoint 실험
- [ ] Day 2 RAG 를 LangGraph 로 재작성 (Agentic RAG 의 본격 버전)

📤 제출

- 그래프 다이어그램 + 코드 + trace 스크린샷