생성형 AI 기반 RAG 개발자 과정

Anthropic 5대 Agentic Pattern

Session 28 / 33 — Day 4

Chaining · Routing · Parallel · Orchestrator · Evaluator

박수현 · (사)한국정보공학기술사회 · 2026

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

이번 시간을 마치면

5대 패턴

- ✅ **Prompt Chaining** — 순차 파이프라인 + gate 검증
- ✅ **Routing** — 분류 후 전문 체인으로 분기
- ✅ **Parallelization** — 팬아웃·팬인, voting, sectioning
- ✅ **Orchestrator-Worker** — 동적 작업 분해 + 위임
- ✅ **Evaluator-Optimizer** — 생성·평가 반복 개선

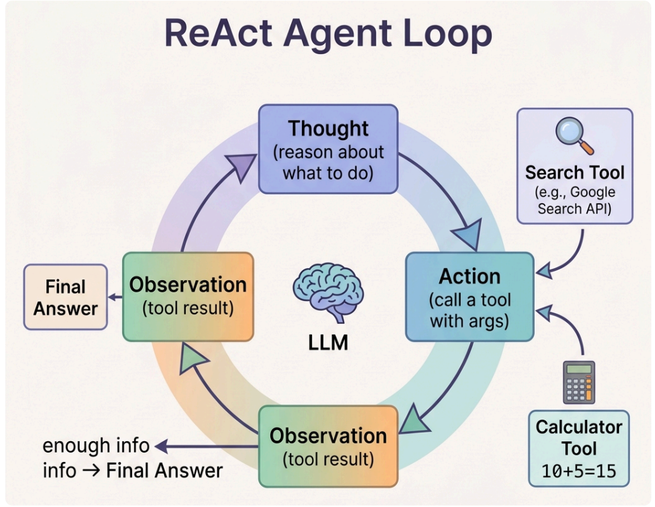
선택과 조합

- ✅ **패턴 선택 결정 트리** + 패턴 **조합** 시나리오
- ✅ **Workflow vs Agent** 의 구분

> 📂 소스: `2.langchain/8.agents/9.agentic_patterns/`

모든 패턴의 기반 — ReAct 루프

에이전트의 **기본 동작**은 단일 루프다 — **생각(Thought) → 행동(Action=도구 호출) → 관찰(Observation)** 을 **답이 충분해질 때까지 반복**한다. Day 4 의 `create_agent` 한 줄이 이 루프를 내부에서 실행한다.



ReAct 루프 · 생각(Thought)→행동(Action=도구 호출)→관찰(Observation)을 반복하다 답이 충분해지면 종료

5대 패턴과의 관계

- **5대 패턴 = 흐름을 프로그래머가 사전 정의** (Workflow)
- **ReAct = LLM 이 루프를 자율 실행** (Agent) → 도구 호출 횟수·순서가 동적

"Workflow" vs "Agent" — 다른 두 개념

Anthropic 정의 (2024)

개념	정의
Workflow	LLM 과 도구가 **사전 정의된 코드 경로** 로 결합
Agent	LLM 이 **자체적으로 흐름·도구 사용을 동적 결정**

두 개념의 사용 가이드

상황	권장
잘 정의된 작업 (예: 번역, 분류, 요약)	**Workflow** (예측 가능, 디버그 쉬움)
입력이 예측 불가능 (예: 자율 에이전트, 코딩 도우미)	**Agent** (유연하지만 비용·복잡도 ↑)
명확한 단계가 있는 복잡 작업	**Workflow** (5대 패턴 중 적합한 것)
도구 사용·횟수가 동적	**Agent** (Day 4 의 ReAct/Tool calling)

"Workflow" vs "Agent" — 원칙과 위치

핵심 원칙

> 1. **단순함 우선**: 한 줄 LLM 호출 → 워크플로우 → 에이전트. 최소 복잡도로 시작.
> 2. **투명성**: Agent 의 사고 과정·도구 호출을 사용자에게 노출 (LangSmith)
> 3. **도구·인터페이스를 신중히 설계**: 모호한 도구 description = 잘못된 호출

5대 패턴 — 모두 Workflow 의 변형

- 5대 패턴은 LLM 의 흐름을 "프로그래머가 정의" 하는 워크플로우
- 본질적 Agent 는 앞 슬라이드의 **ReAct 루프** (생각→행동→관찰) 를 바탕으로 하며, `create_agent` 가 이를 자동 실행한다 (Day 4 Session 25~27)

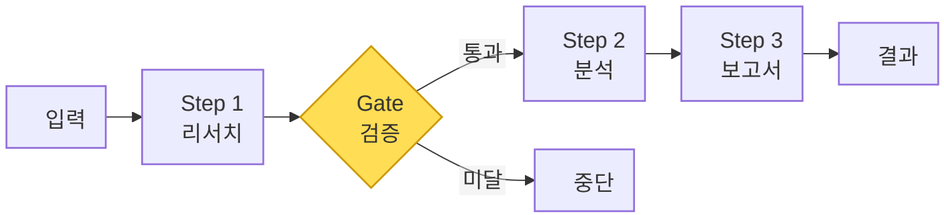
PART A

PART A
패턴 ① — Prompt Chaining

> 순차 파이프라인 + Gate — 약 10분

작업의 순차 단계 분해

구조도



언제 쓰나

- **단계가 명확** 한 작업 (예: 리서치 → 분석 → 보고서)
- Gate 로 **중간 검증** 가능한 경우

장단점

✅	❌
디버그 쉬움	단계 수만큼 latency
각 단계 프롬프트 최적화	분기/반복 불가

코드 — `1.prompt_chaining.py`

```
research = ChatPromptTemplate.from_template(
    "주제: {topic}\n핵심 정보 5개:"
) | llm | parser

# Gate — 충분한가?
def gate(research_text: str) → str:
    if len(research_text) < 200:
        return "INSUFFICIENT"
    return research_text

analyze = ChatPromptTemplate.from_template(
    "리서치 결과:\n{research}\n\n분석 3가지 관점:"
) | llm | parser

report = ChatPromptTemplate.from_template(
    "분석:\n{analysis}\n\n임원용 1페이지 보고서:"
) | llm | parser

# 체이닝
def pipeline(topic: str) → str:
    r = research.invoke({"topic": topic})
    if gate(r) == "INSUFFICIENT":
        return "리서치 부족 — 중단"
    a = analyze.invoke({"research": r})
```

🎯 **핵심**: LCEL 의 `prompt | model | parser` 를 여러 단계로 연결한 구조 — Day 2 Session 9 의 확장이다.

PART B

PART B
패턴 ② — Routing

> 분류 → 전문 체인 분기 — 약 10분

입력 분류 후 전문 체인 분기

언제 쓰나

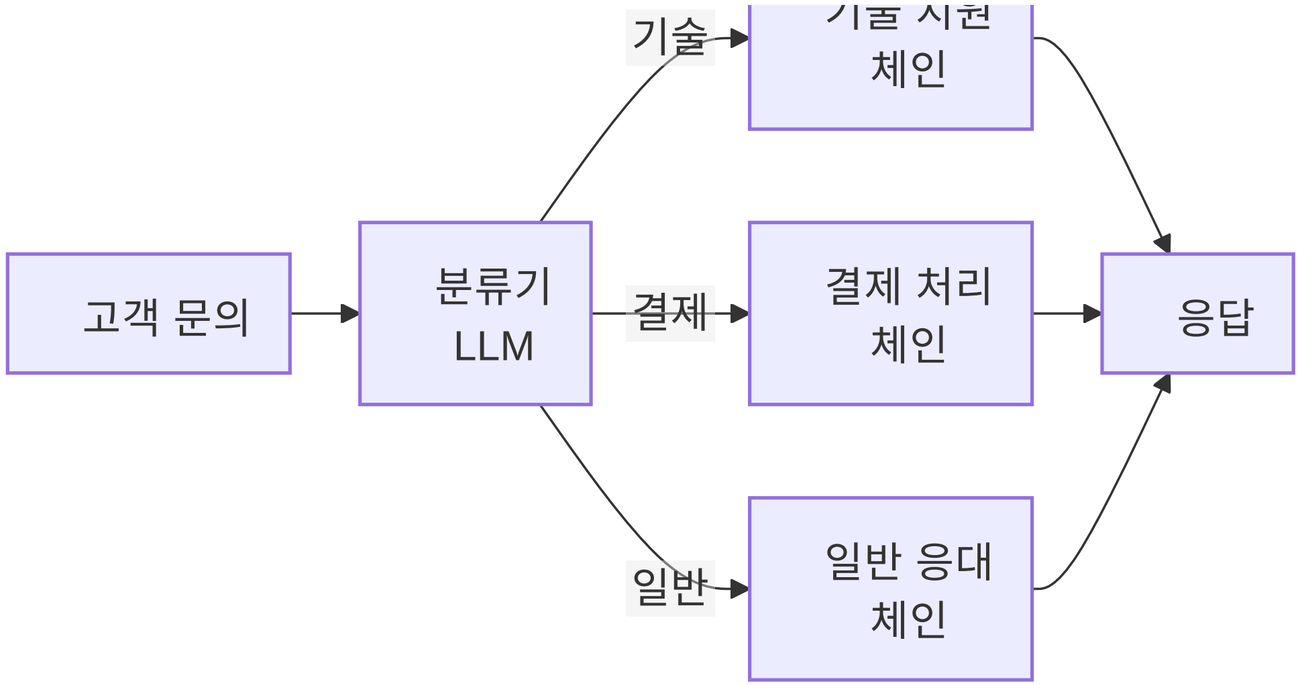
- 입력 종류가 N개로 **명확히 분류** 가능
- 각 부류마다 **다른 처리** 가 적절
- 고객 지원·티켓 라우팅·콘텐츠 모더레이션

장점

- 부류별 system prompt 를 최적화 → 품질 ↑
- 부류별 다른 모델 (저렴/정확) 선택 가능 / 새 부류 추가 시 다른 부류에 영향 X

> 💡 LangGraph 의 conditional edges 로도 동일한 패턴을 구현할 수 있다 (Session 27).

구조도 — Routing (분류 라우팅)



분류기 LLM 이 문의를 N개 부류로 라벨링 → 부류별 **전문 체인** 으로 분기 → 응답.

분류기 + 전문 체인 + 라우터 — `2.routing.py`

```
# === 분류기 — 문의를 N개 부류 중 하나로 라벨링 ===
classifier_prompt = ChatPromptTemplate.from_template(
    "다음 고객 문의를 분류 (technical|billing|general 중 하나만):\n{inquiry}")
classifier = classifier_prompt | llm | parser

# ═══ 전문 체인 3개 (부류별 최적화 prompt) ===
technical = ChatPromptTemplate.from_template(
    "기술 지원 전문가. 코드/에러 메시지로 정확히 답:\n{inquiry}") | llm | parser
billing = ChatPromptTemplate.from_template(
    "결제 전문가. 환불 정책 + 다음 단계 안내:\n{inquiry}") | llm | parser
general = ChatPromptTemplate.from_template(
    "친절한 일반 응대. 짧고 따뜻하게:\n{inquiry}") | llm | parser

# ═══ 라우터 — 분류 라벨에 따라 전문 체인 호출 ===
def route(state: dict) → dict:
    category = classifier.invoke(state).strip().lower()
    if "technical" in category: return technical.invoke(state)
    if "billing"   in category: return billing.invoke(state)
    return general.invoke(state)

print(route({"inquiry": "Python에서 ImportError가 떠요"}))  # → technical 체인이 응답
```

> 🎯 **본질**: 분류 1회 (+1 토큰) 후 부류별로 최적화된 system prompt·모델을 선택 적용한다.

PART C

PART C
패턴 ③ — Parallelization

> Fan-out / Fan-in + Voting — 약 12분

Sectioning vs Voting

변형 1 — Sectioning (작업 분할)

```
하나의 큰 작업 → N개 독립 부분 작업 → 결과 합치기

예: 긴 문서 → 페이지별 요약 (병렬) → 최종 합본
```

변형 2 — Voting (다수결)

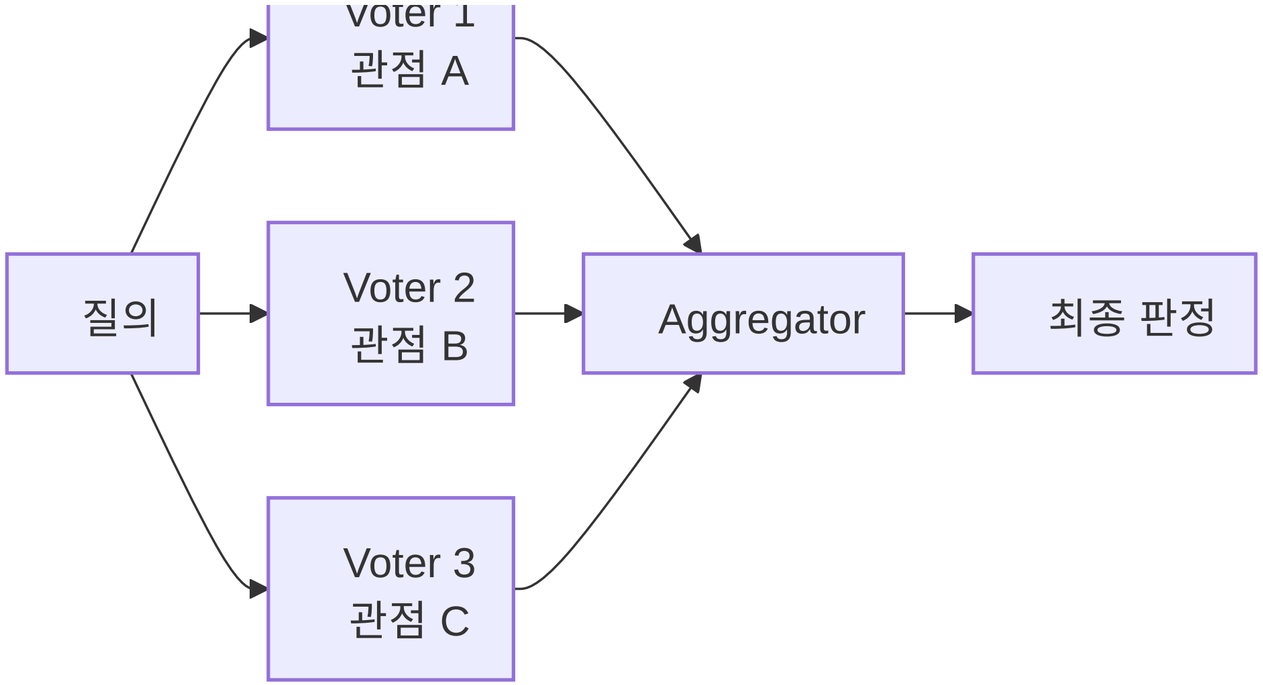
```
하나의 작업 → N번 독립 실행 → 다수결로 최종 선택

예: 코드 보안 검토 → 3개 LLM 독립 평가 → 2개 이상 "위험"이면 차단
```

언제 어느 변형

시나리오	변형
작업이 독립적인 부분으로 분할 가능	**Sectioning**
작업 자체는 하나, 신뢰성이 중요	**Voting**
다양한 관점 종합 필요	Voting

구조도 — Voting (다수결)



동일 질의를 **N명의 voter 가 독립 평가** → Aggregator 가 다수결·평균으로 최종 판정. (Sectioning 은 같은 fan-out 구조지만 각 갈래가 서로 다른 부분 작업을 맡는다.)

`3.parallelization.py` — RunnableParallel

```
from langchain_core.runnables import RunnableParallel

# ═══ 변형 1 — Sectioning (리뷰를 3관점 동시 분석) ═══
quality_prompt    = ChatPromptTemplate.from_template("제품 품질 관점:\n{review}")
service_prompt    = ChatPromptTemplate.from_template("서비스 관점:\n{review}")
delivery_prompt   = ChatPromptTemplate.from_template("배송 관점:\n{review}")

analyze_review = RunnableParallel({
    "quality":  quality_prompt  | llm | parser,
    "service":  service_prompt  | llm | parser,
    "delivery": delivery_prompt | llm | parser,
})

result = analyze_review.invoke({"review": "제품은 좋은데 배송이 너무 늦었어요"})
# → 3개 관점이 병렬 호출 — 전체 latency = max(개별 latency)
print(result["quality"])
print(result["service"])
print(result["delivery"])
```

> ⬇️ 코드가 길어 다음 장에 이어집니다.

`3.parallelization.py` — RunnableParallel (이어서)

```
# === 변형 2 — Voting (번역 평가 3명) ═══
def make_evaluator(persona: str):
    prompt = ChatPromptTemplate.from_template(
        f"당신은 {persona}. 다음 번역의 품질을 1~10 점수로만 출력:\n번역: {{translation}}"
    )
    return prompt | llm | parser

voters = RunnableParallel({
    "linguist":  make_evaluator("언어학자"),
    "native":    make_evaluator("원어민 화자"),
    "domain":    make_evaluator("해당 분야 전문가"),
})

scores = voters.invoke({"translation": "..."})
# → {'linguist': '8', 'native': '9', 'domain': '7'}
avg = sum(int(s) for s in scores.values()) / 3
print(f"평균: {avg:.1f}")
```

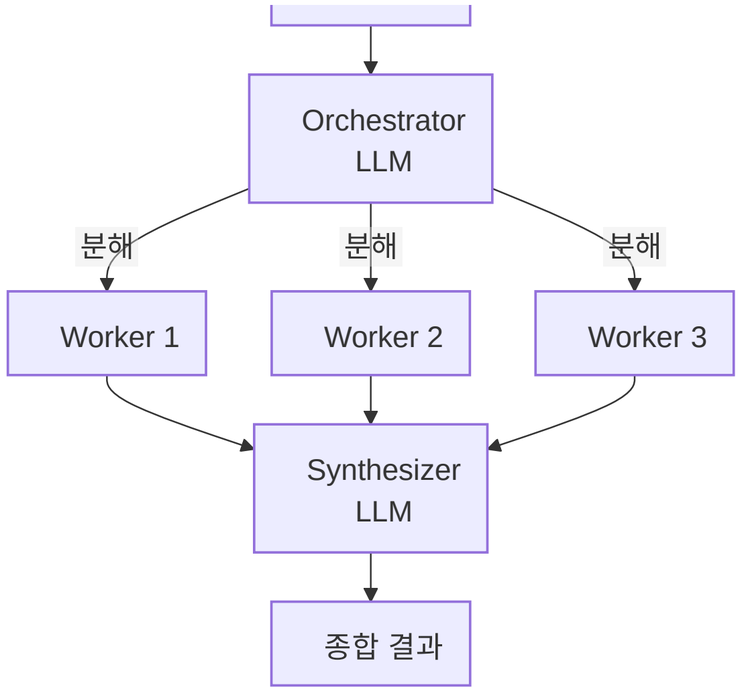
> 🎯 **자동 병렬화** — LangChain LCEL 의 핵심 가치다. `RunnableParallel` 이 async 로 병렬 호출한다.

PART D

PART D
패턴 ④ — Orchestrator-Worker

동적 작업 분해 — 약 15분

"분해는 LLM 이, 처리는 다른 LLM 이"



> Orchestrator LLM 이 입력을 동적으로 sub-task 로 분해 → 각 Worker 가 처리 → Synthesizer LLM 이 종합.

Orchestrator-Worker — 언제 쓰나

Parallelization 과의 차이

Parallelization	Orchestrator
부분 작업이 **사전 정의**	**LLM 이 동적 분해**
항상 같은 수의 worker	입력마다 다름
단순한 fan-out/in	종합 단계도 LLM

언제 쓰나

- 입력 복잡도가 가변적
- 분해 자체가 비자명 (예: "코드 리뷰" — 어떤 측면들이?)
- 작업이 **추론을 요구** 하는 경우

코드 (요약) — `4.orchestrator_worker.py`

```
class State(TypedDict):
    task:      str
    subtasks:  list[str]
    results:   list[dict]
    final:     str

def orchestrate(state: State) → dict:
    """LLM 이 입력을 sub-task 리스트로 분해."""
    prompt = f"""작업: {state['task']}
이를 3~5개 독립 sub-task 로 분해.
JSON 배열: ["sub1", "sub2", ...]"""
    subtasks = json.loads(llm.invoke(prompt).content)
    return {"subtasks": subtasks}

def worker(subtask: str) → dict:
    """각 sub-task 를 처리 (병렬)."""
    return {"subtask": subtask,
            "result": llm.invoke(f"수행: {subtask}").content}
```

⬇️ 코드가 길어 다음 장에 이어집니다.

코드 (요약) — `4.orchestrator_worker.py` (이어서)

```
def synthesize(state: State) → dict:
    """worker 결과들을 종합."""
    bundle = "\n".join(f"{r['subtask']}: {r['result']}" for r in state["results"])
    return {"final": llm.invoke(
        f"다음 sub-task 결과들을 종합:\n{bundle}"
    ).content}

# LangGraph 구성 (간략)
g = StateGraph(State)
g.add_node("orchestrate", orchestrate)
g.add_node("workers",     RunnableLambda(
    lambda s: {"results": [worker(t) for t in s["subtasks"]]}))
g.add_node("synthesize",  synthesize)
g.add_edge(START, "orchestrate")
g.add_edge("orchestrate", "workers")
g.add_edge("workers", "synthesize")
g.add_edge("synthesize", END)
```

> 💡 본 코스 미니프로젝트 (Session 8) 의 RAG 챗봇을 이 패턴으로 확장 가능 — "문서 + 코드 + 차트" 같은 복합 출력.

PART E

PART E
패턴 ⑤ — Evaluator-Optimizer

비평 → 개선 반복 — 약 12분

`5.evaluator_optimizer.py` — Session 27 Self-correction 의 일반화

흐름

```
Generator → 출력
   ↑           ↓
   ↑       Evaluator
   ↑       (점수 + 비평)
   ↑           ↓
   └── 점수 < 임계치? → 비평 반영해 재생성
                 점수 ≥ 임계치? → 최종 출력
```

코드 ① 생성기 (Generator)

```
class State(TypedDict):
    task:     str
    draft:    str
    feedback: str
    score:    float
    attempts: int

def generate(state: State) → dict:
    prompt = f"""작업: {state['task']}
이전 시도: {state.get('draft', '없음')}
평가자 피드백: {state.get('feedback', '없음')}

피드백 반영해 더 나은 결과:"""
    return {"draft":    llm.invoke(prompt).content,
```

5.evaluator_optimizer.py — 평가기 (Evaluator)

코드 ② 평가기 — 점수 + 비평 추출

```
def evaluate(state: State) → dict:
    prompt = f"""작업: {state['task']}
결과: {state['draft']}

점수 (0~10) + 비평:
score: X
feedback: ..."""
    text = llm.invoke(prompt).content
    return {"score": float(re.search(r"score: ([\d.]+)", text).group(1)),
            "feedback": re.search(r"feedback: (.+)", text).group(1)}
```

> 평가기는 생성 결과에 **점수와 비평** 을 함께 매겨, 다음 슬라이드의 루프 종료 조건과 재생성에 쓰인다.

루프 종료 조건 + 그래프 구성

```
def should_continue(state: State) → str:
    if state["score"] ⩾ 8.0:   return END
    if state["attempts"] ⩾ 3:  return END
    return "generate"

g = StateGraph(State)
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_continue, ["generate", END])
app = g.compile()

result = app.invoke({"task": "마케팅 카피 작성: 친환경 텀블러"})
print(f"최종 점수 {result['score']}, 시도 {result['attempts']}")
```

효과

- 코드 생성·번역·카피라이팅 같은 **품질이 중요한 작업** 에서 +10~20%
- 비용·시간: 평균 2~3배 (3번 반복 시)
- 종료 조건은 **점수 임계치(≥8.0)** 와 **최대 시도(3회)** 두 가지 — 무한 루프 방지

PART F

PART F
패턴 선택 기준

> 5가지 패턴 비교 + 결정 트리 — 약 10분

한 페이지 요약

패턴	핵심	적합 작업	토큰 비용
Chaining	순차 + Gate	단계 명확 (리서치 → 분석 → 보고서)	× N
Routing	분류 → 전문가	입력 종류 다양 (고객 지원)	+ 1 (분류)
Parallelization	Fan-out/in	독립 부분 / 다수결	× N (동시)
Orchestrator-Worker	동적 분해	분해 자체가 추론 (코드리뷰)	분해 + N × 처리 + 종합
Evaluator-Optimizer	비평 → 개선	품질이 중요 (카피·코드)	× 2~3 (반복)

결정 트리

```
작업 흐름이 사전에 알려졌나?
├─ Yes → 단계가 N개 직선인가?
│        ├─ Yes → ① Chaining
│        └─ No  → 입력 종류로 분기?
│                 ├─ Yes → ② Routing
│                 └─ No  → 같은 작업 N번?
│                          ├─ 다수결 / 독립 부분 → ③ Parallelization
│                          └─ 분해가 추론 필요 → ④ Orchestrator
│
└─ No  → LLM 이 동적으로 결정?
```

> 🎯 본 코스의 모든 추상화는 결국 이 5패턴 또는 본질적 Agent (ReAct) 로 수렴한다.

패턴 통합 적용 예시

시나리오 1 — "내 도메인 Q&A 봇"

```
사용자 질문
    ↓
[Routing — 일반 대화 vs 도메인 질문]
    ├─ 일반 → 직접 답변
    └─ 도메인 → [Agent + Tools (RAG, Wiki, Calc)] (ReAct)
                    ↓
                최종 답변
```

→ Day 2 RAG + Day 4 Agent + Routing 패턴

시나리오 2 — "리포트 자동 생성"

```
주제 입력
    ↓
[Chaining] 리서치 → Gate → 분석
                              ↓
[Orchestrator-Worker] 분석 결과를 sub-section 별로 분해 → worker 가 각 section 작성
                              ↓
[Evaluator-Optimizer] 평가 → 점수 7 미만 → 재작성 (최대 3회)
                              ↓
                         최종 보고서
```

→ 4가지 패턴 조합

패턴 조합 — 코드 PR 자동 리뷰

시나리오 3 — "코드 PR 자동 리뷰"

```
PR diff 입력
    ↓
[Orchestrator] 5개 측면으로 분해 (보안·성능·스타일·테스트·문서)
    ↓
[Parallelization] 5개 측면 동시 평가 (Voting 형태)
    ↓
[Routing] 위험도 high → 사람 알림 / low → 자동 코멘트
```

→ 3가지 패턴 조합

> 💡 실전에서는 **단일 패턴보다 조합** 이 일반적이다. 실제 작업에서 적합한 조합을 판단하는 감각은 반복 적용으로 형성된다.

핵심 7가지

1. ✅ **Workflow(정해진 경로)** 와 **Agent(자율 결정)** 는 다른 개념
2. ✅ **ReAct 루프** — 생각→행동(도구)→관찰 반복, 모든 Agent 의 바탕 (`create_agent`)
3. ✅ **Prompt Chaining** — 작업을 단계로 분할하고 게이트로 검증
4. ✅ **Routing** — 입력을 분류해 전문 분기로
5. ✅ **Parallelization** — Sectioning / Voting 으로 병렬 처리
6. ✅ **Orchestrator-Worker** — 분해는 LLM, 처리는 워커
7. ✅ **Evaluator-Optimizer** — 생성 → 평가 → 재시도 루프