

## Day 3

# LangGraph & Multi-Agent Systems

상태 기반 워크플로우와 에이전트 협업

# Agent란 무엇인가?

자율적으로 작동하는 지능형 시스템



Agent는 **환경을 인식**하고, **사고**하며, **행동**하여  
목표를 달성하는 자율적인 시스템입니다



자율성  
(Autonomy)

스스로 판단하고 실행

- 사용자 개입 최소화
- 상황에 맞는 결정
- 목표 지향적 행동

예: 여행 Agent가 예산과  
취향을 고려해 자동으로  
호텔을 선택



도구 사용  
(Tool Use)

외부 도구와 API 활용

- 검색 엔진
- 데이터베이스 쿼리
- API 호출

예: 날씨 API로 기온 확인,  
지도 API로 거리 계산,  
예약 API로 호텔 예약



피드백 루프  
(Feedback Loop)

결과를 보고 재시도

- 실행 결과 관찰
- 오류 시 재시도
- 전략 수정

예: 호텔 검색 실패 시  
다른 지역이나 가격대로  
재검색

# LLM vs RAG vs Agent

단계별 진화

## LLM

단발성 대화

### 질문:

"서울 여행 추천해줘"

### 답변:

"경복궁, 남산타워,  
명동 쇼핑..."

### 특징:

- ✗ 학습 데이터만 활용
- ✗ 최신 정보 없음
- ✗ 1회성 응답

### 한계:

- 오래된 정보
- 활각 가능성
- 개인화 불가

## RAG

검색 + 생성

### 질문:

"서울 여행 추천해줘"



최신 여행 가이드 문서

### 답변:

"최신 여행지 문서에 따르면  
경복궁은 화-일 운영..."

### 특징:

- ✓ 문서 기반 답변
- ✓ 최신 정보 반영
- 여전히 1회성

### 개선점:

- ✓ 사실 기반 답변
- ✗ 실시간 행동 불가

## Agent

자율 실행

### 질문:

"서울 여행 추천해줘"



1. 날씨 API 호출 ☀
2. 영업시간 확인 ⏳
3. 리뷰 검색 ★
4. 거리 계산 🚗
5. 최적 루트 생성 ✨

"맑은 날씨 예상, 경복궁  
화-일 10-18시 운영..."

### 특징:

- ✓ 자율적 도구 사용
- ✓ 실시간 정보
- ✓ 반복 실행 가능

🚀 가장 강력한 시스템!

# Agent 4대 핵심 구성요소

Perception → Reasoning → Action → Memory

## Perception

환경 관찰 & 데이터 수집

- 사용자 입력 이해
- 외부 상태 인식

## Reasoning

판단 & 계획 수립

- 목표 분석
- 전략 결정

## Action

도구 실행 & 외부 작업

- API 호출
- 데이터 조작

## Memory

상태 저장 & 학습

- 대화 이력
- 중간 결과

# 실전 예시: 여행 Agent

서울 3박4일 여행 계획

1

## 感知 (인식)

사용자: "예산 100만원으로 서울 3박4일 여행 계획 짜줘"



2

## 추론 (Reasoning)

- 예산 100만원 분석 → 숙박 40만원, 식비 30만원, 교통 15만원, 관광 15만원
- 필요한 정보: 날씨, 영업시간, 호텔 가격, 교통편, 맛집 리뷰  
→ 계획: 날씨 확인 → 호텔 검색 → 교통 계획 → 일정 생성



3

## 행동 (Action)

- 날씨 API 호출: "서울 다음주 맑음, 평균 18°C"
- 호텔 API 검색: "명동 호텔 3박 35만원"
- 지도 API 거리 계산: "경복궁 → 북촌 2.1km"
- 리뷰 DB 검색: "광장시장 맛집 Top 5"



4

## 저장 (Memory)

검색 결과 저장 → 다음 단계 활용

## 최종 결과

상세한 3박4일 일정 + 예산 내역

# LangGraph란?

상태 기반 Agent 워크플로우 프레임워크

## 핵심 아이디어

Agent를 그래프로 표현하고 상태를 전달하며 워크플로우 실행



노드 간 공유 데이터

- TypedDict로 정의
- 노드마다 업데이트



실제 작업 수행 함수

- State 받아 처리
- LLM 호출, API 실행



노드 간 연결 & 흐름

- 일반 Edge: 항상 이동
- Conditional Edge: 조건 분기

# State 정의 이론

TypedDict + Annotated + Reducer

## State 구조

### 기본 형태

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

class State(TypedDict):
```

### 일반 필드

```
destination: str
budget: int
days: int
# 덮어쓰기 방식
```

### Reducer 필드

```
messages: Annotated[
    list, add_messages
]
# 병합 방식 ✨
```

# State 예제: 여행 Agent

실제 코드로 보기

```
class TravelState (TypedDict):
    destination : str      # "서울"
    budget       : int      # 1000000
    days         : int      # 3
    weather      : str      # "맑음, 18°C"
    hotels       : list     # [...]
    plan         : str      # 최종 일정
    messages     : Annotated [
        list , add_messages
    ]      # 대화 이력 자동 병합 ✨
```

# Graph 구축 이론

StateGraph + Node + Edge + Compile

## 🔧 Graph 구축 4단계

### 1 StateGraph 생성

```
graph = StateGraph(TravelState)
```

### 2 Node 추가

```
graph.add_node("search", search_hotels)
```

### 3 Edge 연결

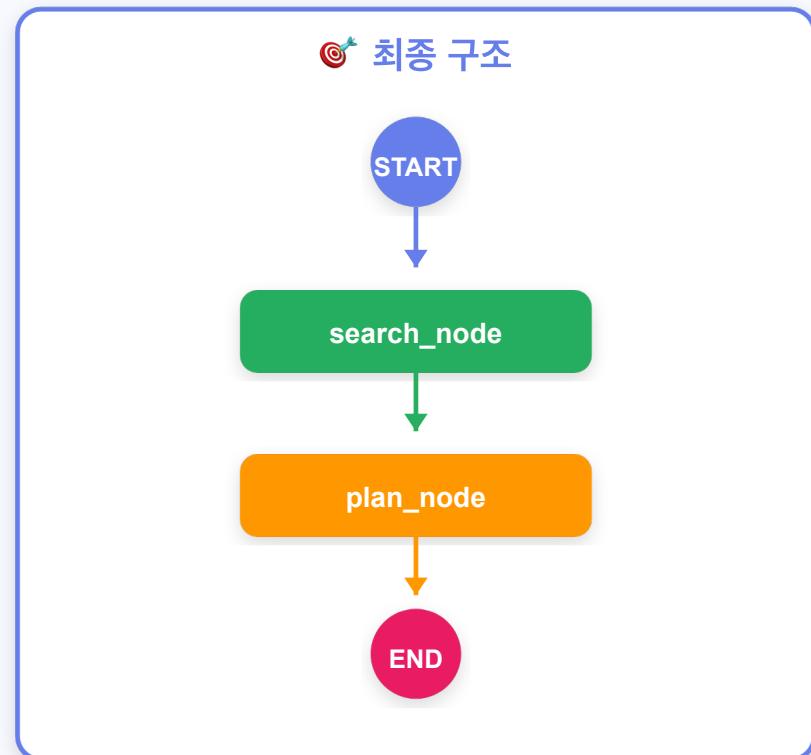
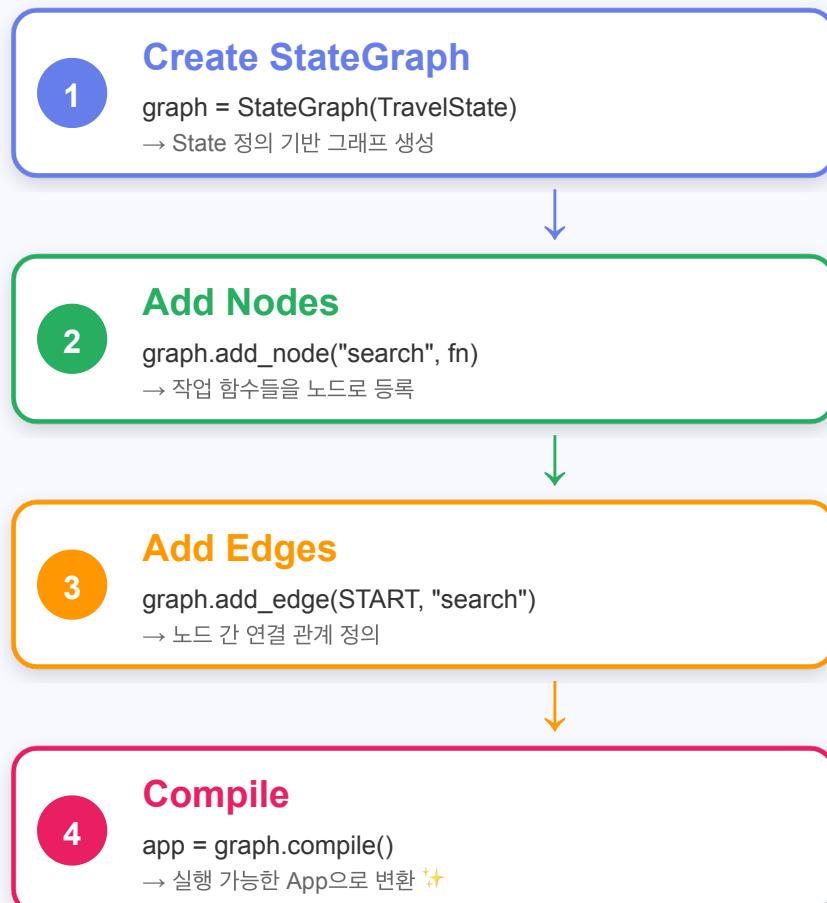
```
graph.add_edge("search", "book")
```

### 4 Compile 실행

```
app = graph.compile()
```

# StateGraph 컴파일 4단계

그래프를 실행 가능한 App으로 변환



## Graph 예제: 여행 추천



```
graph LR; input["input"] --> search["search"]; search --> output["output"]  
graph TD; input["input"] --- process_input["process_input"]; search["search"] --- search_hotels["search_hotels"]; output["output"] --- generate_result["generate_result"]  
graph TD; input["input"] --> search["search"]
```

graph .add\_node( "input" , process\_input)  
graph .add\_node( "search" , search\_hotels)  
graph .add\_node( "output" , generate\_result)  
graph .add\_edge( "input" , "search" )

# Command Pattern

동적으로 다음 노드 선택 + State 업데이트

## 💡 Command란?

노드가 다음 노드를 직접 지정하고, 동시에 State도 업데이트

### goto: 다음 노드

어디로 갈지 결정

```
Command(goto="next_node")
```

### update: 상태 변경

State에 무엇을 추가할지

```
update={"result": data}
```

## 📌 실전 예시

호텔이 있으면 → 예약 노드로

```
Command(goto="book",
update={"hotels": [...]})
```

호텔이 없으면 → 재검색 노드로

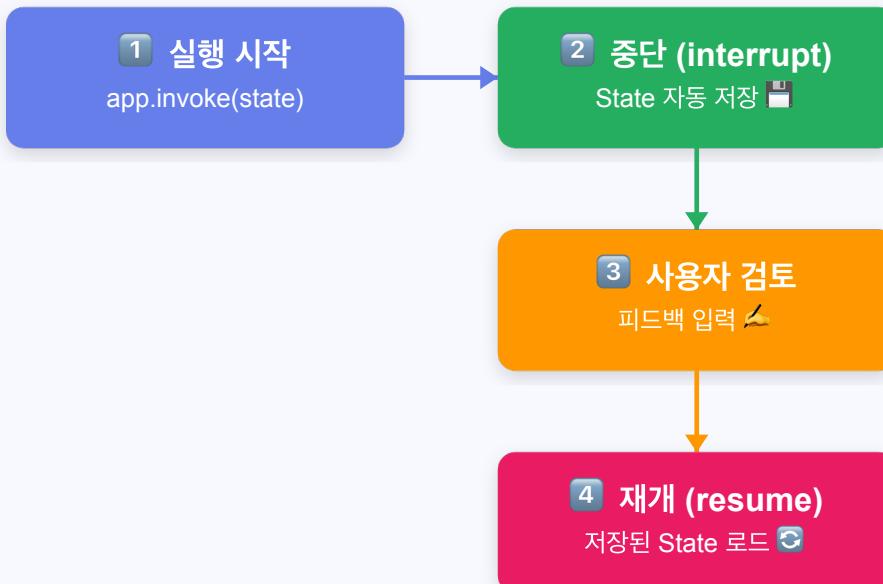
```
Command(goto="retry",
update={"error": "..."})
```

# Checkpointing

실행 중 State를 저장하고 나중에 이어서 실행

## 💡 왜 필요한가?

Human-in-the-Loop: 사용자 검토 후 다시 실행

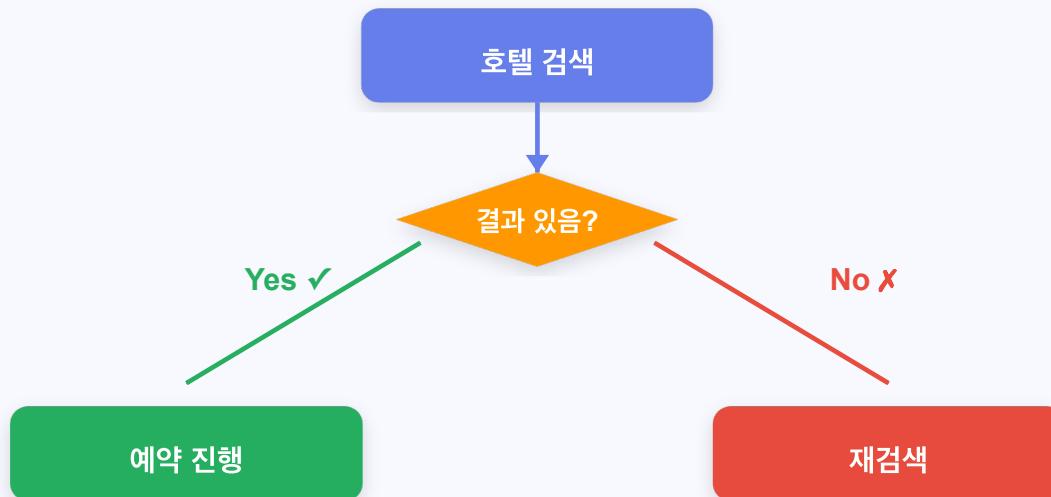


## 💻 코드 예시

```
checkpointer = MemorySaver() | app = graph.compile(checkpointer=checkpointer)
```

# Conditional Edge

조건부 라우팅 - 상황에 따라 다른 경로로 분기



## 💻 Conditional Edge 구현

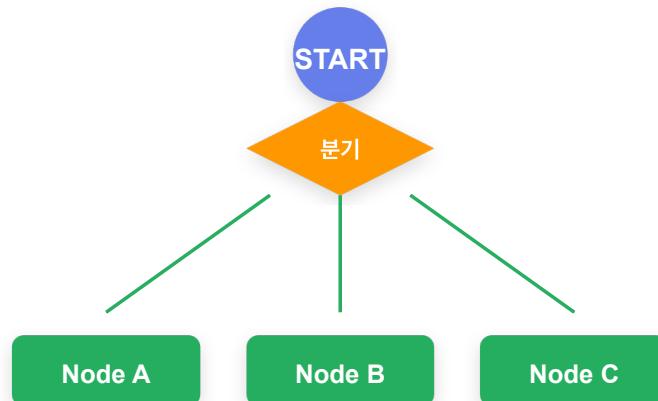
```
def router (state):
    if state == [ "hotels" ]:
        return "book"      # 예약 노드로
    return "retry"       # 재검색 노드로
```

# Conditional Edge 두 가지 방식

시작점 분기 vs 중간 분기

## 1 Entry Point 분기

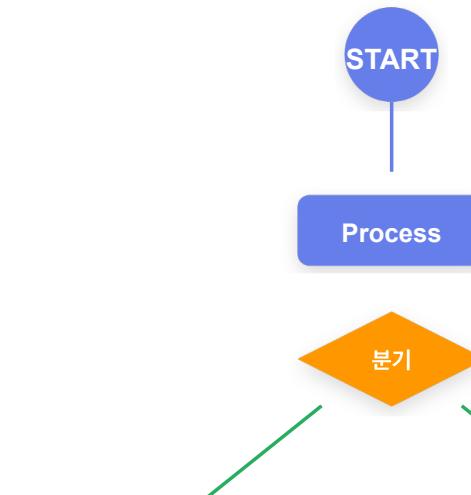
START 직후 바로 분기



```
graph. set_conditional_entry_point
  router_fn ,
  { "A" : "node_a" , ... }
```

## 2 중간 노드 분기

특정 노드 실행 후 분기



```
graph. add_conditional_edges
  "process" , decide_fn ,
  { "X" : "node_x" , ... }
```

# Human-in-the-Loop (HITL)

사용자 피드백을 워크플로우에 통합

## 💡 핵심 개념

Agent가 중요한 결정 전에 **멈춰서** 사용자 승인을 받고  
피드백을 반영하여 **재개**합니다



### ⏸ interrupt()

#### 동작:

1. 현재 State 저장
2. 사용자에게 정보 전달
3. 피드백 대기

### ▶ resume()

#### 동작:

1. 사용자 피드백 수신
2. State 업데이트
3. 다음 노드로 진행



**핵심 포인트:** interrupt()로 멈추고 사용자 의견을 받아 resume()으로 재개

# HITL 실전: 여행 계획 승인

Before/After 비교

## ✗ HITL 없음

### Agent 동작:

1. 호텔 검색 → 자동 예약
2. 일정 생성 → 바로 확정
3. 사용자 확인 없음

### 문제점:

- 잘못된 예약 가능성
- 사용자 선호도 무시
- 수정 불가
- 비용 낭비 위험

## ✓ HITL 적용

### Agent 동작:

1. 호텔 검색 완료
2. 사용자: "2번 호텔로 변경"
3. 수정된 호텔로 예약

### 장점:

- ✓ 사용자 통제권 유지
- ✓ 실시간 수정 가능
- ✓ 만족도 향상

# ReAct Agent 이론

Reasoning + Acting

## ReAct 루프

### 1. Observe

상황 관찰

### 2. Think

다음 행동 결정



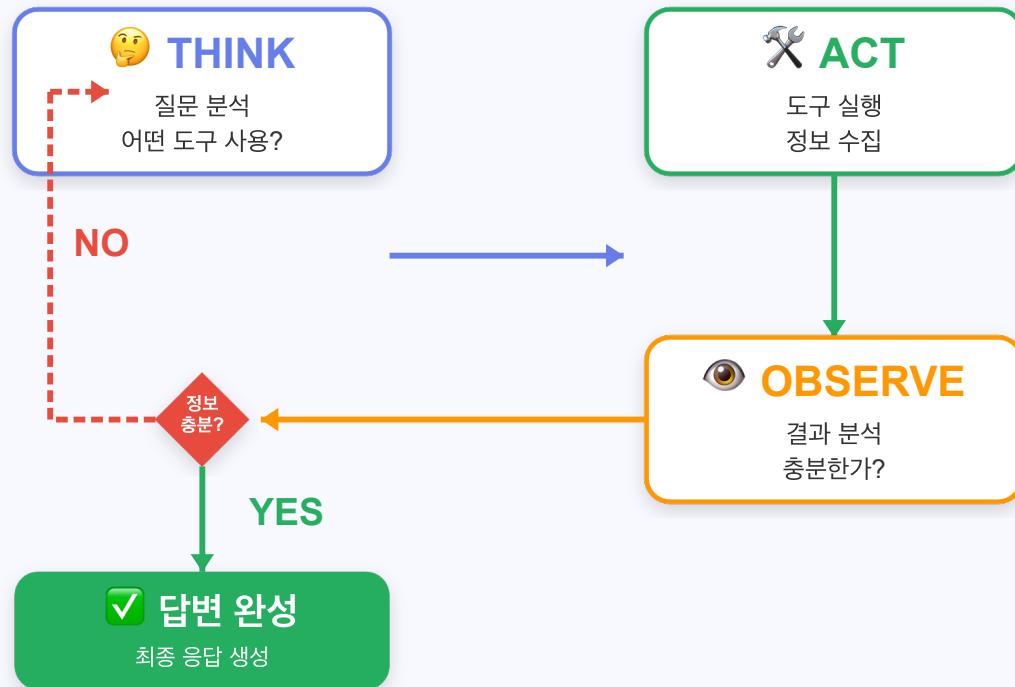
### 3. Act

도구 실행

☞ 목표 달성까지 반복

# ReAct 3단계 상세

Think → Act → Observe 반복



📌 예시: "서울 날씨는?"

1 THINK: "날씨 API 사용하자"

2 ACT: weather\_api.call("서울") → "맑음, 18°C"

3 OBSERVE: "충분함! 답변 가능"

# Tool Registry

Agent가 사용할 도구 정의

## 🛠 Tool 정의 3단계

### 1 함수 작성

```
def get_weather (city: str) -> str :  
    return f" {city} 날씨는 맑음"
```

### 2 Tool로 등록

```
tools = [  
    get_weather , search_hotels ]
```

### 3 LLM에 바인딩

```
llm_with_tools = llm .bind_tools(tools)
```

# Tool Binding 방식

Manual vs Auto - 어떻게 도구를 연결하나?

## 1 Manual (텍스트 파싱)

### ✓ 장점

- 모든 LLM 지원 (EXAONE, Llama, GPT 등)
- Tool Calling 기능 불필요
- 완전한 제어 가능
- 로컬 모델도 OK
- 안정적이고 예측 가능

### 동작 원리

#### 1 LLM이 텍스트로 판단

"날씨를 알려면 weather\_api 사용"

#### 2 개발자가 파싱

```
if "weather" in decision:  
    call_weather_api()
```

#### 3 결과 반환

## 2 Auto (bind\_tools)

### ⚠️ 제약

- OpenAI / Anthropic만 지원
- Tool Calling 기능 필수
- API 비용 발생
- LLM 판단 실패 시 문제  
→ 편리하지만 블랙박스

### 동작 원리

#### 1 도구 자동 등록

```
model.bind_tools([weather_api])
```

#### 2 LLM이 JSON 생성

```
{"tool": "weather", "args": {...}}
```

#### 3 자동 실행 & 결과 반환

# ReAct 실전: 여행 정보 수집

## 1. Observe: "서울 날씨 확인 필요"

상태: weather = None

## 2. Think: "get\_weather 도구 호출"

LLM 결정: use\_tool("get\_weather", city="서울")

## 3. Act: API 실행

결과: "서울: 맑음, 18°C"

## 4. Update: State 업데이트

상태: weather = "맑음, 18°C"

→ 다음 작업(호텔 검색)으로 진행

# Multi-Agent 이론

## 여러 Agent가 협업

순차 협업

Agent 1



Agent 2



Agent 3

병렬 처리

Agent A

Agent B



Merge

# HandOffs 패턴

Agent 간 상태 전달하기

## 💡 HandOffs란?

한 Agent가 작업을 완료하면 다음 Agent에게 결과를 전달  
→ 복잡한 작업을 전문화된 Agent들이 순차적으로 처리

### Agent 1

숙박 전문가

- 작업: 호텔 검색
- 결과: 3곳 추천



### Agent 2

교통 전문가

- 작업: 이동 경로
- 결과: 지하철+버스



### Agent 3

맛집 전문가

- 작업: 맛집 추천
- 결과: 5곳 선정

## 코드 예시: Agent 간 전달

```
def hotel_agent(state):
    hotels = search_hotels(state[      "destination"  ])
    return { "hotels" : hotels}        # Agent 2로 전달

graph.add_edge(      "hotel_agent" , "transport_agent" )
```

# Fan-out / Fan-in 패턴

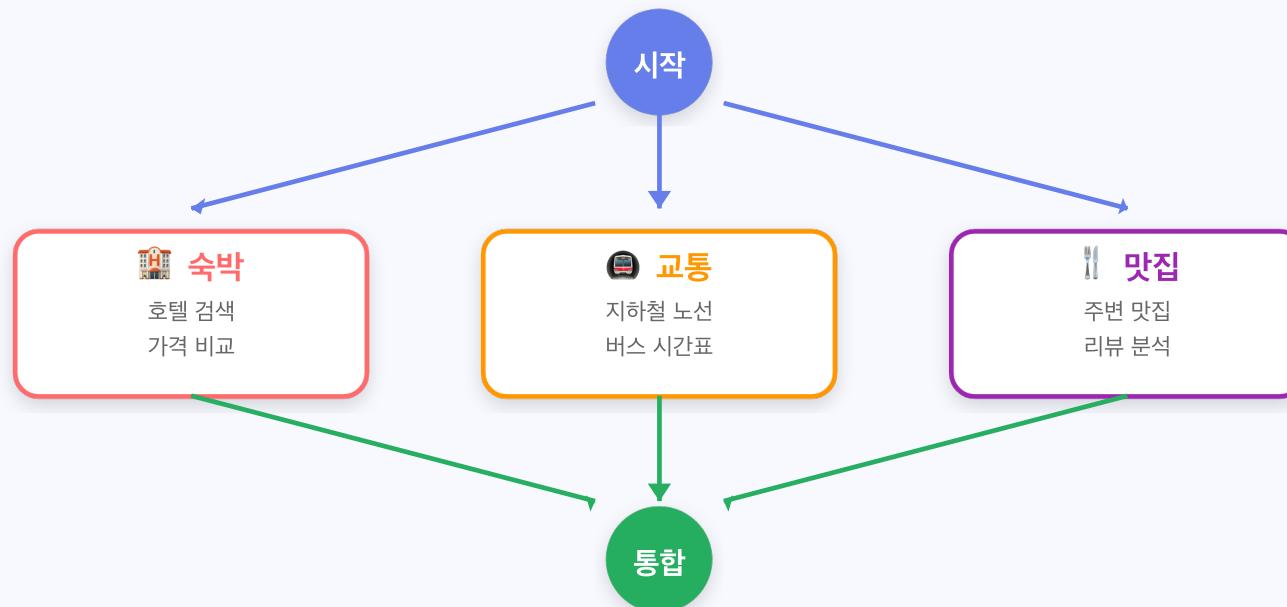
병렬 처리로 속도 높이기

## ⭐ Fan-out

하나의 작업을 여러 Agent에 동시 분배

## 🎯 Fan-in

여러 Agent의 결과를 하나로 통합



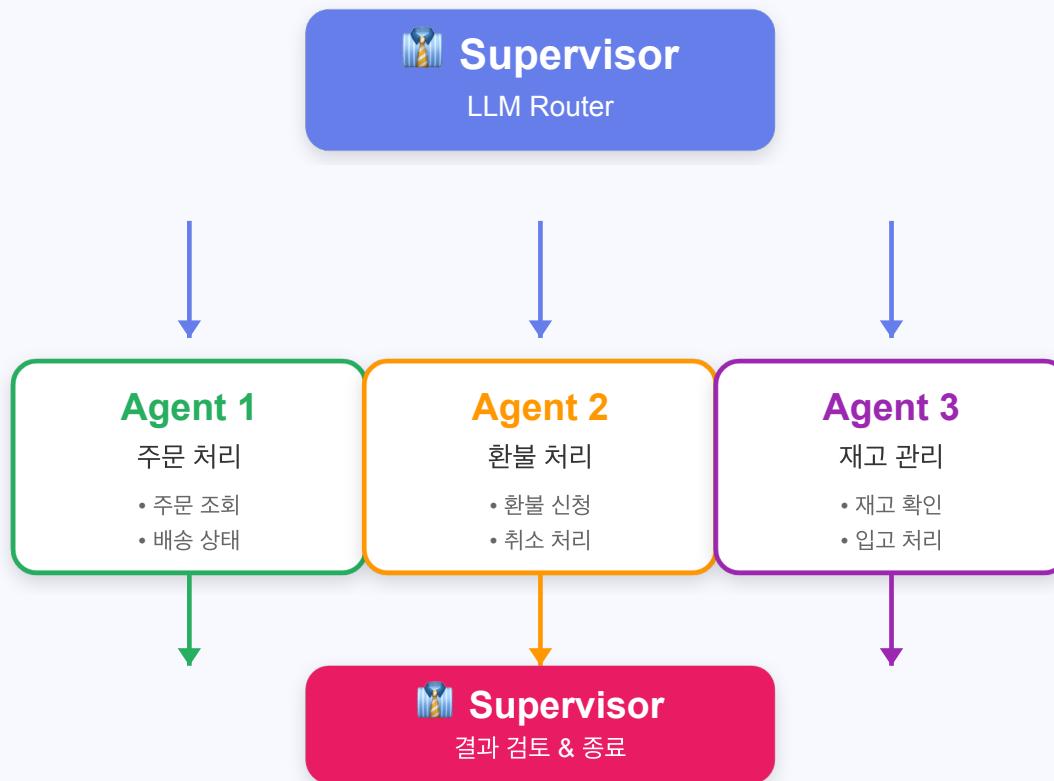
## 💡 병렬 처리의 장점

1 속도: 순차 처리 대비 3배 빠름

2 독립성: 각 Agent 실패해도 다른 Agent 영향 없음

# Supervisor Pattern

중앙 관리자가 작업을 분배하고 결과를 취합



특징

✓ 역할 분담 명확

✓ 품질 관리 용이

✓ 확장 쉬움

# Hierarchical Pattern

2단계 위계: 전체 관리자 → 팀 관리자 → 작업자

## 전체 Supervisor

총괄 관리

### 리서치 팀

#### 팀 Supervisor

품질 검토

#### Agent 1

웹 검색

#### Agent 2

논문 분석

### 콘텐츠 팀

#### 팀 Supervisor

스타일 검토

#### Agent 1

초안 작성

#### Agent 2

편집

## 작업 흐름

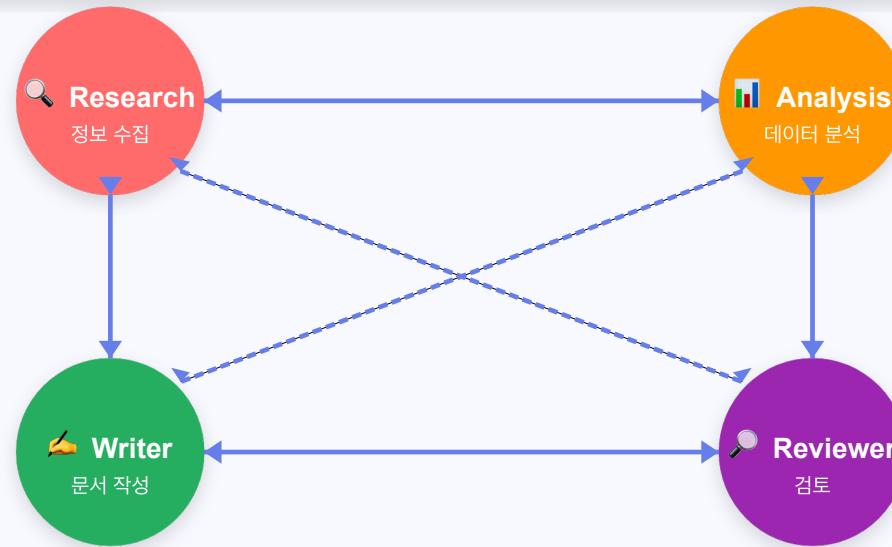
전체 Supervisor → 팀 선택 → 팀 Supervisor → 작업자 배정 → 팀 검토 → 전체 Supervisor

# Network 패턴

Agent들이 자유롭게 메시지를 주고받는 구조

## 핵심 개념

중앙 컨트롤러 없이 Agent들이 **직접 통신**하여  
필요한 정보를 교환하고 협업



## 특징

- 1 유연성: 필요에 따라 자유롭게 통신
- 3 분산 처리: 병목 현상 최소화

- 2 확장성: Agent 추가/제거 용이

- 복잡도: 통신 경로 관리 필요

# Multi-Agent 실전 예제

서울 여행 계획 팀

## 👤 사용자 요청

"3일간 서울 여행, 예산 50만원, 맛집 위주로 추천해줘"

## ◉ Coordinator Agent: 요청 분석 & 작업 분배

### 숙박 Agent

#### 작업:

- 예산: 15만원 (30%)
- 위치: 명동/종로 중심
- 결과: 게스트하우스 3곳

### 맛집 Agent

#### 작업:

- 예산: 25만원 (50%)
- 조건: 4.5★ 이상
- 결과: 한식/양식 10곳

### 교통 Agent

#### 작업:

- 예산: 10만원 (20%)
- 경로: 최적 동선 계획
- 결과: 3일 이동 계획

## ✓ 통합 결과: 3일 완벽 여행 계획서

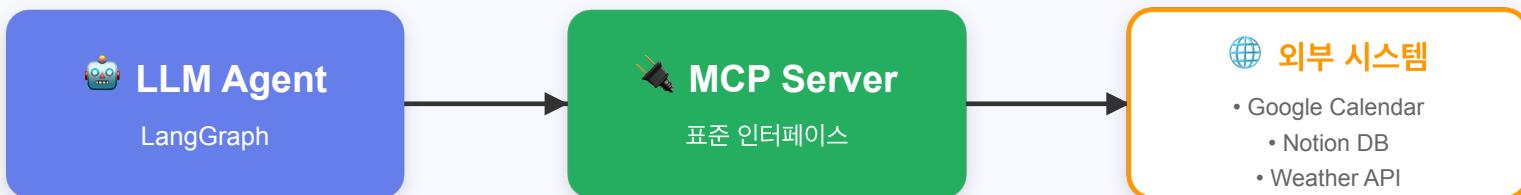
숙소 3곳 + 맛집 10곳 + 최적 동선 + 실시간 업데이트

# MCP (Model Context Protocol)

외부 시스템과 연결하기

## 💡 MCP란?

LLM이 외부 데이터/도구에 안전하게 접근할 수 있도록 하는 표준 프로토콜  
→ API, DB, 파일 시스템, 브라우저 등 다양한 리소스 연결



## 📌 MCP 활용 사례

### 1 여행 Agent

- 날씨 API로 실시간 날씨
- Google Maps로 거리 계산

### 2 업무 Agent

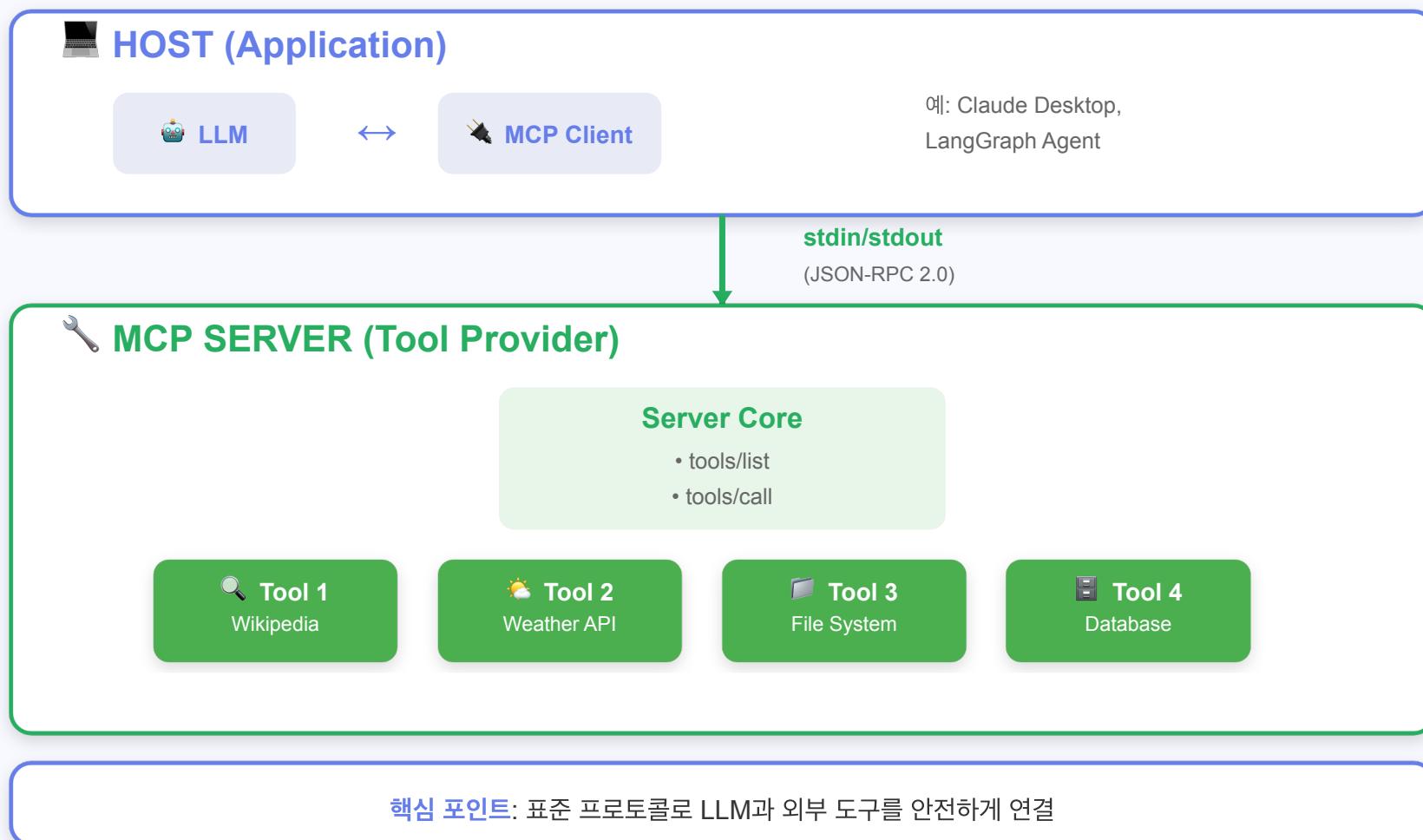
- Notion에서 TODO 읽기
- Slack으로 알림 전송

### 3 데이터 Agent

- PostgreSQL 쿼리 실행
- CSV/Excel 파일 읽기

# MCP 3-Tier 구조

Host ↔ Client ↔ Server



# JSON-RPC 2.0 프로토콜

표준화된 요청/응답 구조

## 1 tools/list

사용 가능한 도구 목록 조회



```
{
  "jsonrpc" : "2.0" ,
  "id"      : 1 ,
  "method"   : "tools/list"
}
```



```
{
  "jsonrpc" : "2.0" ,
  "id"      : 1 ,
  "result"  : {
    "tools" : [
      {
        "name" : "search" , ...
      }
    ]
  }
}
```

## 2 tools/call

특정 도구 실행



```
{
  "jsonrpc" : "2.0" ,
  "id"      : 2 ,
  "method"   : "tools/call" ,
  "params"   : {
    "name"   : "search" ,
    "arguments": {
      "query" : "Python"
    }
  }
}
```



```
{
  "jsonrpc" : "2.0" ,
  "id"      : 2 ,
  "result"  : {
    "content" : "Python is..."
  }
}
```

# Tool Definition 포맷

JSON Schema로 도구 정의



## Tool Definition 예시

```
{  
  "name" : "search_wikipedia" ,  
  "description" : "Search Wikipedia" ,  
  "inputSchema" : {  
    "type" : "object" ,  
    "properties" : {  
      "topic" : {  
        "type" : "string" ,  
        "description" : "Search topic"  
      },  
      "sentences" : [{ "type" : "integer" , ... }]  
    },  
    "required" : [ "topic" ]  
  }  
}
```

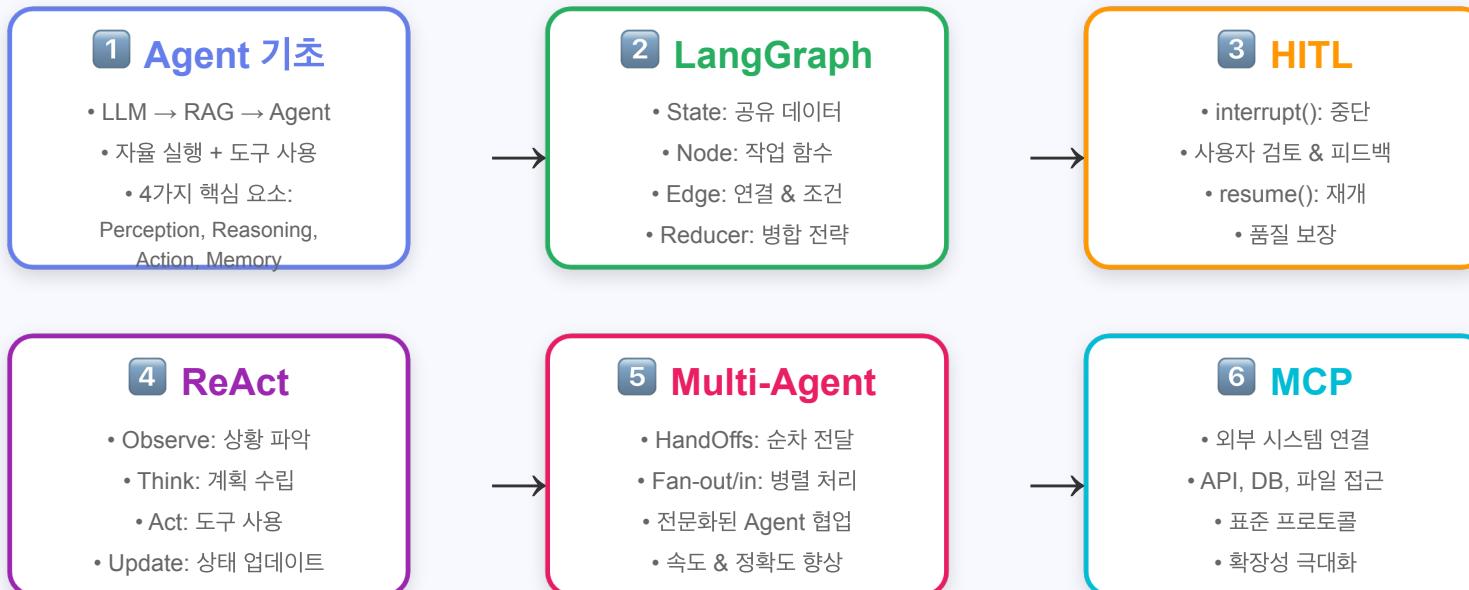
1 name: 도구 이름

2 description: 설명

3 inputSchema: 파라미터 정의

# 전체 요약

Day 3: LangGraph & Multi-Agent Systems



## 🎯 핵심 메시지

LangGraph로 복잡한 Agent 시스템을 구조화하고  
Multi-Agent 협업으로 실전 문제를 효율적으로 해결!

## 실무 적용 가이드

언제 어떤 패턴을 사용할까?

### 🤔 어떤 문제를 해결하려고 하나요?

작업의 특성에 따라 적합한 패턴을 선택하세요

#### 단순 자동화

→ 기본 ReAct Agent

- ✓ 문서 요약
- ✓ 단순 정보 검색
- ✓ 이메일 자동 응답

#### 품질 중요

→ HITL 패턴

- ✓ 고객 응대 메시지
- ✓ 중요 보고서 작성
- ✓ 의사결정 지원

#### 복잡한 작업

→ Multi-Agent

- ✓ 여행 계획 (숙박+교통+맛집)
- ✓ 시장 분석 (뉴스+주가+SNS)
- ✓ 프로젝트 관리 (일정+리소스)

### 실전 체크리스트



State 설계: 필요한 데이터만 State에 포함



도구 선택: 신뢰할 수 있는 API/DB만 연결



에러 처리: 실패 시나리오 항상 고려



테스트: 작은 예제로 먼저 검증



로깅: 디버깅을 위한 상세 로그 남기기



비용: LLM 호출 횟수 최적화



Day 3 완료! 여러분의 Agent를 만들어보세요!