

# Reasoning

---

ReAct

# Agent Overview

---

- ReAct, PlanExec : Agent의 사고 루프
- Tool Calling: Agent의 행동 팔
- RAG: Agent의 지식 눈
- Memory: Agent의 기억 뇌

# ReAct

- 논문: **Synergizing reasoning and acting in language models** (2022.10 arXiv 공개, 저자: Princeton University와 Google Research 연구원들)

- 핵심 개념: **Reason(추론) + Act(행동)**

- LLM이 단순히 텍스트를 생성하는 것을 넘어, 생각(Thought)을 하고 그 생각을 바탕으로 행동(Action)을 취하며, 그 결과 (Observation)를 다시 피드백 받아 다음 생각을 이어 가는 순환 구조의 추론 방식

## 1. Reason(추론):

- LLM이 내부적으로 작업 계획을 세우고, 진행 상황을 추적하며, 전략을 수정하는 능력
- 이는 주로 Chain-of-Thought (CoT) 프롬프팅에서 영감을 받았음
- 예: "질문에 답하려면 먼저 A를 검색하고, 그다음 B를 검색해야겠다."

## 2. Act(행동):

- LLM이 외부 환경이나 도구와 상호작용하는 능력
- 예: 위키피디아 검색 API 호출, 계산기 사용, 코드 실행 등.

# Hotspot QA - 복잡한 사실 위주 질문

## (1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

### (1a) Standard

Answer: iPod



### (1b) CoT (Reason Only)

Thought: Let's think step by step. Apple Remote was originally designed to interact with Apple TV. Apple TV can be controlled by iPhone, iPad, and iPod Touch. So the answer is iPhone, iPad, and iPod Touch.

Answer: iPhone, iPad, iPod Touch



### (1c) Act-Only

Act 1: Search[Apple Remote]  
Obs 1: The Apple Remote is a remote control ...

Act 2: Search[Front Row]  
Obs 2: Could not find [Front Row]. Similar: ...

Act 3: Search[Front Row (software)]

Obs 3: Front Row is a discontinued media center software ...

Act 4: Finish[yes]



## (1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.

Act 1: Search[Apple Remote]

Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple ... originally designed to control the Front Row media center program ...

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.

Act 2: Search[Front Row]

Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (software)', ...]

Thought 3: Front Row is not found. I need to search Front Row (software).

Act 3: Search[Front Row (software)]

Obs 3: Front Row is a discontinued media center software ...

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.

Act 4: Finish[keyboard function keys]



# AlfWorld 데이터셋 - 상호 작용 환경

## (2) AlfWorld

You are in the middle of a room. Looking quickly around you, you see a cabinet 6, a cabinet 1, a coffee machine 1, a countertop 3, a stove burner 1, and a toaster 1.  
Your task is to: Put some pepper shaker on a drawer.

## (2a) Act-Only

Act 1: Go to drawer 1  
Obs 1: The drawer 1 is closed.

Act 2: Open drawer 1  
Obs 2: You open the drawer 1. The drawer 1 is open. In it, you see a dishponge 2, and a spoon 1.

Act 3: Go to sinkbasin 1  
Obs 3: On the sinkbasin 1, you see a dishponge 3, a spatula 1, and a spoon 2.

Act 4: Take peppershaker 1 from sinkbasin 1  
Obs 4: Nothing happens.

Act 5: Take peppershaker 1 from sinkbasin 1  
Obs 5: Nothing happens.



## (2b) ReAct (Reason + Act)

Act 1: Think[First I need to find a pepper shaker...more likely to appear in cabinets (1-6), countertops (1-3), ... ]

Act 2: Go to cabinet 1

Obs 2: On the cabinet 1, you see a vase 2.

(Here the agent go to cabinet 1, then cabinet 2, 3, then countertop 1 and 2)

Act 6: Go to countertop 3

Obs 6: On the countertop 3, you see a apple 1, a bread 1, ..... a pepper shaker 1, and a vase 3.

Act 7: Take pepper shaker 1 from countertop 3

Obs 7: You pick up the pepper shaker 1 from the countertop 3.

Act 8: Think[Now I find a pepper shaker 1. Next, I need to put it in/on drawer 1.]

Act 9: Go to drawer 1

Obs 9: Drawer 1 is closed.

Act 10: Open drawer 1

Obs 10: You open Drawer 1 ...

Act 11: Put pepper shaker 1 in/on drawer 1

Obs 11: You put pepper shaker 1 in/on the drawer 1.



# ReAct의 중요성

- **기존 방식의 한계 극복**
  - Chain-of-Thought(CoT)의 한계인 추론은 잘하지만 외부의 최신 정보에 접근하지 못해서 hallucination 생성하는 문제점 해소
  - Act-Only의 한계인 복잡한 계획 수립이나 문제 해결 능력 부족 극복
- 사실기반 질의응답(HotpotQA)이나 복잡한 상호작용이 필요한 작업(ALFWorld)에서 **기존 방식들보다 우수한 성능 보임**
  - HotpotQA – prompting만으로 하는 것보다 CoT나 fine-tuning으로 학습하는 것이 성적이 좋았음.
  - ALFWorld – 단 1~2개의 예시만 보고도 수천 개의 데이터로 훈련된 모방학습이나 강화학습을 압도하였음.
  - 현재(2025년): ReAct 논문이 나온 지 3년이 지난 지금, 상황이 역전됨
  - 당시 PaLM 모델보다 훨씬 강력한 GPT-4, Claude 3, Gemini 같은 최신 모델들은 Fine-tuning 없이도 ReAct의 Thought-Act-Observation 포맷을 훨씬 더 잘 따르고, 요즘은 ReAct를 Fine-tuning하기보다, 강력한 모델에 'Zero-shot' 또는 'Few-shot' 프롬프트로 ReAct 패턴을 지시하는 것이 표준적인 에이전트 구현 방식이 되었음.
- **해석 가능성(Interpretability)**: 모델이 어떤 생각을 바탕으로 어떤 행동을 했는지 추론과정을 추적할 수 있기 때문에 모델의 결정에 대하여 이해하기 쉽다.

# ReAct 프롬프트 예

---

당신은 질문에 답하는 AI 어시스턴트입니다.

당신은 복잡한 질문에 답하기 위해 단계별로 \*\*'생각(Thought)'\*\*하고 \*\*'행동(Act)'\*\*할 수 있습니다.

사용 가능한 도구:

Search[검색어]: 최신 정보를 검색합니다.

Calculator[계산식]: 수학 계산을 수행합니다.

답변은 다음 형식을 엄격하게 따라야 합니다:

**Thought:** 현재 상황을 분석하고 다음 행동을 계획합니다.

**Act:** 사용할 도구와 입력값을 지정합니다. (예: Search[오늘 서울 날씨])

**Observation:** 도구 사용 결과를 받습니다. (이 부분은 시스템이 실제로 채워줍니다)

... (최종 답을 찾을 때까지 Thought/Act/Observation 반복)

...Thought: 모든 정보를 수집했으며, 최종 답변을 구성할 수 있습니다. Final Answer: 사용자에게 제공할 최종 답변입니다.

# ReAct 구현하기 (Text 기반 react agent)

# Python 구현 구조 설계

## A. 데이터 모델

- Trace: 한 턴의 {thought, action, observation}를 기록
- Trajectory: 전체 턴 목록 + 종료 시 final\_answer

## B. 도구(Action) 레이어

- Tool 프로토콜(이름, 설명, 입력 스키마, run(\*\*kwargs)), 등록 레지스트리(dict)
- 예: search\_web(query), calculator(expr), get\_weather(city) 등

## C. 프롬프트 템플릿(Prompt Template)

- 시스템 규칙 + 사용자 질문 + “이력(Thought/Action/Observation)” + 포맷 지시
- 모델이 반드시 포맷을 지키도록 유도:

Thought: ...

Action: <tool\_name>[json\_args]

또는 최종:

Final: <final\_answer>

# Python 구현 구조 설계

## D. 파서(Parser)

- 모델 출력 문자열에서
  - Final: 이면 종료/답변
  - Action: name[json] 패턴이면 name과 args를 추출
- 불일치/형식오류 방지: 정규표현식 + JSON 검증

## E. 컨트롤러(Agent Loop)

- 입력(질문) → 반복:
  - 프롬프트 생성 → LLM 호출
  - 파싱 → Action이면 도구 실행 → Observation 추가
  - Trace 축적, 중단 조건 검사(max\_steps, 토큰 한도, 금지 도구 등)
  - Final을 받으면 Trajectory 종료

## F. 메모리/스토리지(선택)

- 짧은 문제는 에피소드 메모리(현 세션 이력)로 충분
- 장기/지식기반 필요 시 벡터DB/RAG로 Observation을 확장

# Trace and Trajectory

```
@dataclass
class Trace:
    thought: Optional[str] = None
    action_name: Optional[str] = None
    action_args: Optional[Dict[str, Any]] = None
    observation: Optional[str] = None
    raw_model_output: Optional[str] = None
```

```
@dataclass
class Trajectory:
    question: str
    steps: List[Trace] = field(default_factory=list)
    final_answer: Optional[str] = None
```

# Agent Loop

```
for _ in range(max_steps):
```

- ① build prompt with trajectory and ReAct prompt
- ② request query with prompt to llm
- ③ parsed = parse llm output
- ④ make an instance of trace = Trace(thought=parsed.thought,  
raw\_model\_output=llm\_out)
- ⑤ Final이면 종료: check if parsed.final: return trajectory.append(trace)
- ⑥ Action이면 도구 실행하고 도구 결과를 Trace로 만들어 trajectory에 append

# Prompt Construction

```
return PROMPT_TEMPLATE.format(  
    system_rules=SYSTEM_RULES,  
    tool_specs=tools.specs_for_prompt(),  
    question=question,  
    history=history,  
    format_guide=FORMAT_GUIDE,  
)
```

- Agent Loop안에서 반복 시행
- history는 trajectory 내용을 string으로 만들어줌

# history ← trajectory

```
if not traj.steps:  
    history = "(no previous steps)"  
else:  
    lines = []  
    for i, s in enumerate(traj.steps, 1):  
        if s.thought:  
            lines.append(f"Thought: {s.thought}")  
        if s.action_name:  
            lines.append(f"Action: {s.action_name}[{json.dumps(s.action_args or {},  
ensure_ascii=False)}]")  
        if s.observation is not None:  
            lines.append(f"Observation: {s.observation}")  
    history = "\n".join(lines)
```

# System Rules

---

**SYSTEM\_RULES** = """You are a helpful ReAct-style assistant.

Follow the instructions **\*strictly\***.

Your behavior rules:

- You must think step-by-step using the Thought → Action → Observation → Final format.
- The available tools are listed below. You may call only these tools.
- After receiving an Observation, use that information directly to decide your next step.
- Do not repeat the same tool twice for the same question.
- When you can produce the final answer, write 'Final:' and stop.
- Output ONLY the next Thought and either an Action or a Final, nothing else.

"""

# Format Guide

---

```
FORMAT_GUIDE = """
```

Your output must be in one of the following two forms:

1. When you need to use a tool:

Thought: <your reasoning in one sentence>

Action: <tool\_name>[<json\_args>]

여기서 <json\_args>를 감싸는 대괄호 '[' 는 필수입니다.

2. When you can answer directly:

Thought: <your reasoning in one sentence>

Observation: <result from the previous Action>

Final: <your final answer>

```
"""
```

# Prompt Template

---

```
PROMPT_TEMPLATE = """{system_rules}
```

Available tools:

```
{tool_specs}
```

Conversation so far:

User: {question}

```
{history}
```

Follow the format strictly.

```
{format_guide}
```

```
"""
```

# Parser

```
# ===== Parser (Thought / Action / Final) =====
```

```
THOUGHT_RE = re.compile(r"Thought:\s*(.*?)(?:\n|$)", re.DOTALL)
FINAL_RE = re.compile(r"Final:\s*(.*)", re.DOTALL)
ACTION_RE = re.compile(
    r"[Aa]ction\s*: \s*([a-zA-Z0-9_\-]+)\s*(?:\[([^\{\}\]*\})\])",
    re.DOTALL,
)
```

# Tool

```
from typing import Protocol, Dict, Any
```

```
class Tool(Protocol):
```

```
    name: str
```

```
    description: str
```

```
    schema: Dict[str, Any]
```

```
def run(self, **kwargs) -> str: ...
```

- static duck type → 새가 오리처럼 걷고 꽹꽥 소리내면 나는 그 새를 오리라고 부르겠다. Protocol은 상속 관계에 얹매이지 않고 오직 클래스의 \*\*구조(메서드와 속성)\*\*만을 기준으로 타입을 정의할 수 있게 해주는 강력한 타입 힌팅 도구입니다.

# CalculatorTool

```
class CalculatorTool:  
    name = "calculator"  
    description = "간단한 파이썬 수식을 계산합니다. 예: {'expr': '2+3*4'}"  
    schema = {  
        "type": "object",  
        "properties": {"expr": {"type": "string"}},  
        "required": ["expr"],  
    }  
  
    def run(self, **kwargs) -> str:  
        expr = kwargs["expr"]  
        try:  
            val = eval(expr, {"__builtins__": {}}, {})  
            return str(val)  
        except Exception as e:  
            return f"(calculator error: {e})"
```

# ToolRegistry

```
class ToolRegistry:
    def __init__(self) -> None:
        self._tools: Dict[str, Tool] = {}

    def register(self, tool: Tool) -> None:
        self._tools[tool.name] = tool

    def get(self, name: str) -> Tool:
        if name not in self._tools:
            raise KeyError(f"Unknown tool: {name}")
        return self._tools[name]

    def specs_for_prompt(self) -> str:
        """프롬프트에 보여줄 툴 설명 텍스트."""
        lines = []
        for t in self._tools.values():
            lines.append(
                f"- {t.name}: {t.description} "
                f"params_schema={json.dumps(t.schema, ensure_ascii=False)}"
            )
        return "\n".join(lines)
```