WWW.DEFORMATIC.AI.KR · 유민수 개발자

OPENAI AGENTS SDK

에이전트 런타임 완전 이해 가이드

AI 사용자 · 개발자 · 솔루션 기획자 · 기업 AX 담당자를 위한
처음부터 실전까지, 70장 기술 교육 덱

이 덱에서 다루는 것

- OpenAI Agents SDK의 전체 구조와 핵심 개념
- Agent, Runner, Tools, Handoffs, Guardrails, Sessions, Tracing
- 실전 설계 패턴과 안티패턴
- Python 코드 예시와 운영 관점 해설

핵심 메시지 4가지

1. Agents SDK는 에이전트 **런타임**이다
2. 핵심은 Agent, Runner, Tools, Handoffs, Guardrails, Sessions, Tracing
3. 설계 포인트는 실행 제어, 상태 관리, 도구 통합, 운영 가시성
4. 단일 에이전트 설계를 먼저 잘 하는 것이 중요하다

도입

왜 지금 AGENT SDK를 봐야 하는가

2023년까지 대부분의 AI 개발은 "프롬프트를 잘 쓰면 된다"는 수준에서 멈췄습니다. 그러나 현업에서 실제로 동작하는 AI 시스템을 만들어본 개발자와 기획자들은 공통된 벽을 경험합니다. 모델 하나에 프롬프트 하나를 붙이는 방식으로는 비즈니스 요구사항을 충족하기 어렵다는 것입니다.

기존 방식의 한계

- 단일 LLM 호출로는 복잡한 업무 처리 불가
- 외부 시스템 연동 시 직접 코드 구현 필요
- 대화 상태를 직접 관리해야 함
- 오류 발생 시 추적/재현이 어려움
- 멀티 스텝 워크플로우 구현 난이도 높음

AGENTS SDK가 제공하는 것

- 모델 호출 루프를 런타임 수준에서 제어
- 도구 등록과 실행을 SDK가 자동 처리
- 세션과 대화 상태를 구조화된 방식으로 관리
- 트레이싱으로 실행 흐름 완전 가시화
- 가드레일과 HITL로 통제 가능한 자동화

ⓘ 2024~2025년, 기업 AX 담당자와 개발자에게 가장 필요한 역량은 "모델을 잘 호출하는 것"이 아니라 "에이전트 시스템을 설계하고 운영하는 것"으로 이동하고 있습니다. Agents SDK는 그 설계의 언어입니다.

도입

PROMPT APP과 AGENT RUNTIME의 차이

많은 사람들이 "AI 앱을 만든다"고 할 때 실제로는 프롬프트 앱을 만드는 경우가 대부분입니다. 두 가지는 구조적으로 다른 시스템입니다.

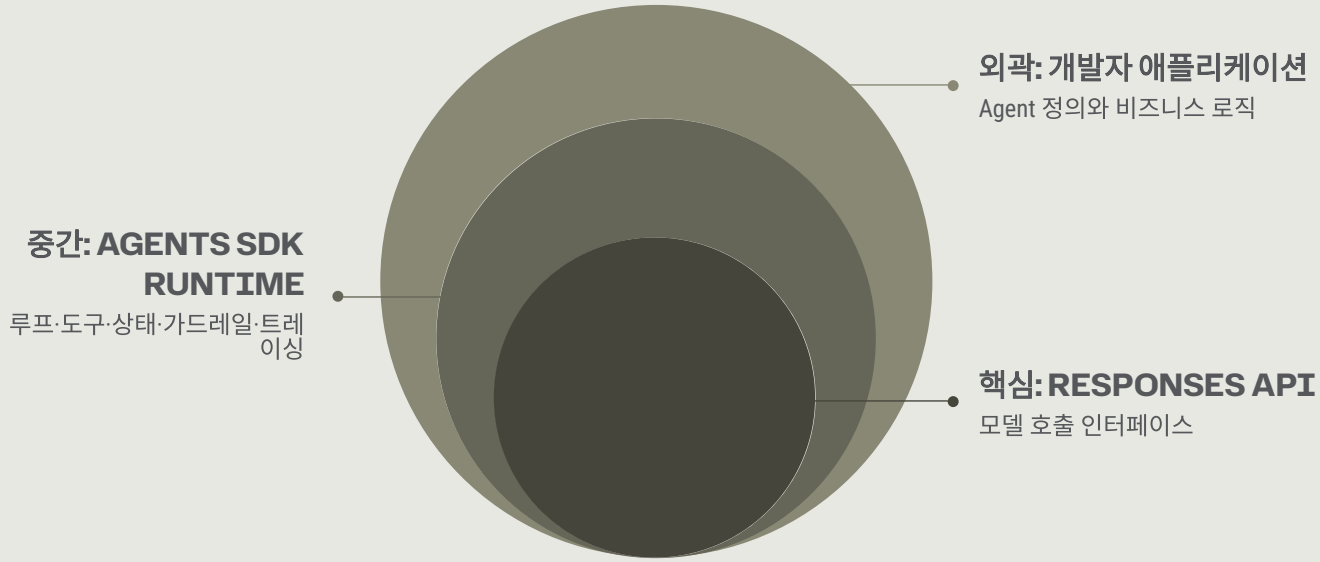
구분	Prompt App	Agent Runtime
실행 구조	사용자 입력 → 모델 호출 → 출력 (1회)	입력 → 루프(모델+도구) → 최종 출력 (N회)
상태 관리	없거나 외부에서 직접 관리	RunState, Session으로 구조화
도구 연동	직접 API 호출 코드 작성	SDK가 tool call 파싱 및 실행 처리
오류 처리	직접 try/except	Guardrail, tripwire, pause/resume
가시성	로그 직접 구성	Trace/Span 자동 생성
멀티 에이전트	직접 구현	Handoff 메커니즘 내장

핵심 차이는 **"루프를 누가 제어하는가"**입니다. 프롬프트 앱은 개발자가 루프를 구현합니다. Agent Runtime은 SDK가 루프를 소유하고, 개발자는 루프 안의 동작을 정의합니다.

도입

RESPONSES API와 AGENTS SDK의 관계

OpenAI의 API 체계를 이해해야 Agents SDK의 위치가 명확해집니다. Agents SDK는 Responses API 위에 올라타는 에이전트 오케스트레이션 계층입니다.



RESPONSES API란

OpenAI가 2024년 출시한 새로운 모델 호출 API입니다. Chat Completions API와 달리 built-in 도구(웹 검색, 파일 검색 등)를 네이티브로 지원하며, 대화 상태를 서버 측에서 관리하는 `previous_response_id` 방식을 제공합니다.

AGENTS SDK가 추가하는 것

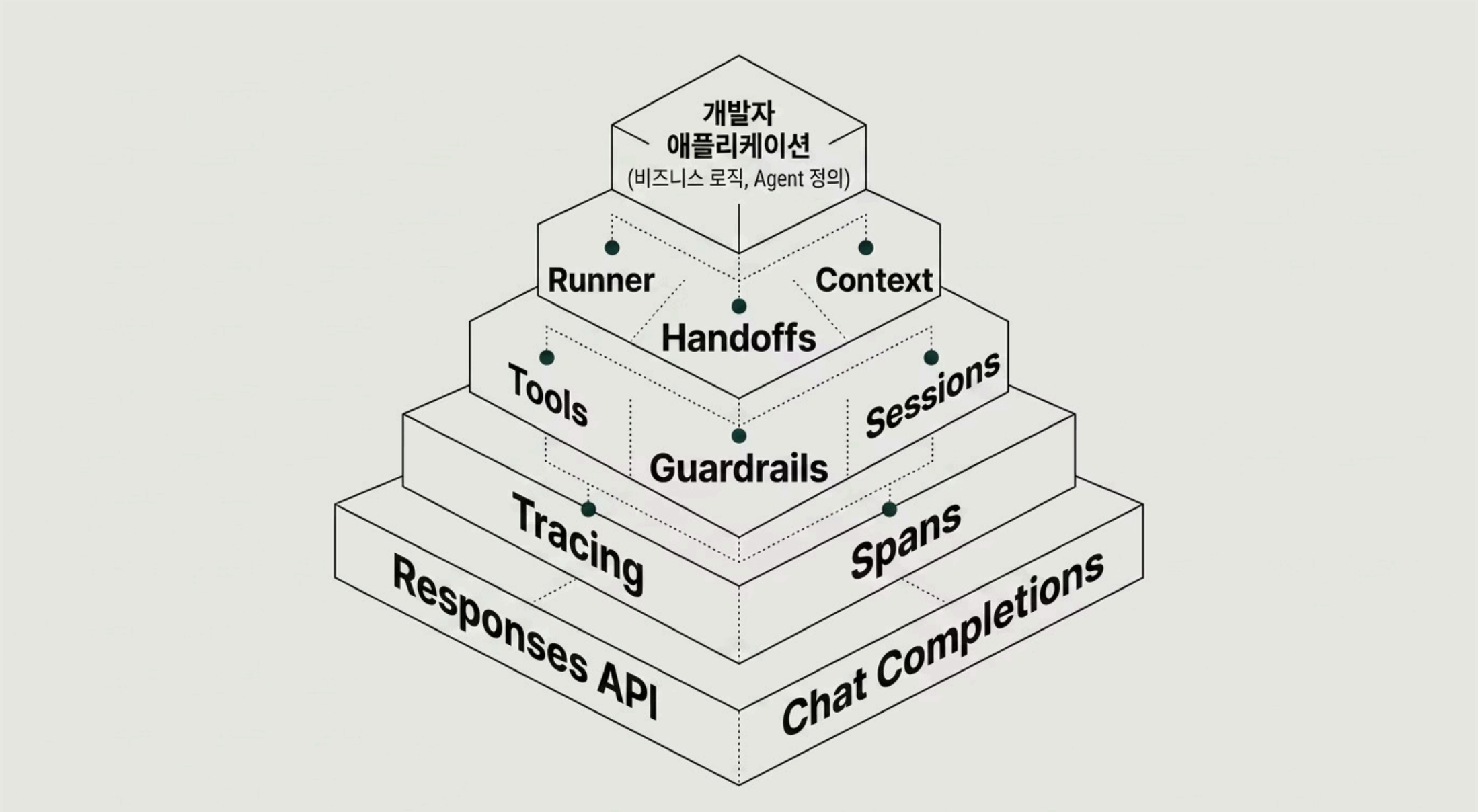
Responses API는 모델을 호출하는 인터페이스입니다. Agents SDK는 그 위에서 에이전트 루프 실행, 도구 파싱과 호출, 핸드오프 처리, 가드레일 적용, 트레이싱 수집을 담당합니다. 즉, SDK는 모델 API를 감싸는 런타임 계층입니다.

Agents SDK는 Chat Completions API 경로도 지원합니다. 모델 계층을 추상화하여 백엔드를 교체 가능하게 설계되어 있습니다. 하지만 기본 권장 경로는 Responses API입니다.

도입

AGENTS SDK 전체 구조 한눈에 보기

Agents SDK를 구성하는 핵심 레이어와 컴포넌트를 계층 구조로 파악합니다.



수직 분리의 의미

각 계층은 서로 다른 관심사를 담당합니다. 모델 인터페이스 계층은 API 호출만 담당하고, 오케스트레이션 계층은 루프 실행과 흐름 제어만 담당합니다. 이 분리가 유지보수성과 테스트 가능성을 높입니다.

어느 계층에서 문제가 생기는가

실제 개발에서 가장 많은 문제는 오케스트레이션 계층과 도구·제어 계층에서 발생합니다. 모델이 잘못 호출되는 것보다 루프 설계, 도구 스키마, 가드레일 미흡이 더 흔한 실패 원인입니다.

도입

핵심 개념 7가지 미리보기

이 덱 전체에서 반복적으로 등장하는 7가지 핵심 개념을 먼저 간략하게 소개합니다. 각 개념은 이후 섹션에서 상세히 다룹니다.

1	**AGENT** 이름, 지시사항, 모델, 도구, 핸드오프를 가진 실행 단위. LLM 호출의 주체이자 행동의 주체.
2	**RUNNER** Agent를 실행하는 루프 엔진. 입력을 받아 모델 호출 → 도구 실행 → 판정을 반복하며 최종 결과를 반환.
3	**TOOLS** Agent가 호출할 수 있는 외부 기능. Python 함수, Hosted 도구, 로컬 실행 도구, 다른 Agent를 포함.
4	**HANDOFFS** 한 Agent가 다른 Agent로 대화 제어를 이전하는 메커니즘. 멀티에이전트 오케스트레이션의 핵심.
5	**GUARDRAILS** 입력/출력/도구 호출을 검증하는 안전 계층. 위반 시 실행을 차단하거나 사람의 승인을 요청.
6	**SESSIONS** 대화 상태를 구조화된 방식으로 저장하고 불러오는 메커니즘. 세션 저장소와 연결 가능.
7	**TRACING** 에이전트 실행의 모든 단계를 Trace/Span으로 기록. 디버깅, 평가, 운영 가시성의 핵심 도구.

도입

이 덱을 어떻게 읽으면 좋은가

이 덱은 70장 규모의 기술 교육 자료입니다. 독자의 목적과 배경에 따라 다른 읽기 경로를 추천합니다.

독자 유형	권장 읽기 경로	특히 주목할 섹션
처음 접하는 AI 사용자	1장부터 순서대로	도입, A(큰 그림), M(코드 예시)
Python 개발자	도입 → B → D → M	B(아키텍처), D(도구), M(코드 예시)
솔루션 기획자	도입 → A → E → L	A(큰 그림), E(멀티에이전트), L(실전 설계)
기업 AX 담당자	도입 → A → H → K → L	H(Guardrails), K(운영), L(실전 설계), N(마무리)

각 섹션 마지막에는 **"실무 포인트"** 또는 **"한 줄 정리"**가 있습니다. 빠르게 훑어볼 때는 이 요약 박스만 읽어도 핵심을 파악할 수 있습니다.

⚠️ 이 덱이 전제하는 것

Python 기초 문법을 이해하고 있으며, OpenAI API를 한 번 이상 사용해본 경험이 있다고 가정합니다. 완전 입문자라면 OpenAI 공식 튜토리얼을 먼저 보고 오는 것을 권장합니다.

A. 큰 그림 이해

큰 그림 이해

AGENT SDK의 문제의식부터 시작하기

단순 LLM 호출의 한계 · 루프와 상태의 필요성 · SDK의 범위

A. 큰 그림 이해

단순 LLM 호출의 한계: 왜 루프가 필요한가

가장 기본적인 LLM 호출은 입력 → 출력의 1회 구조입니다. 간단한 질의응답에는 충분하지만, 현업의 실제 태스크는 거의 항상 이 구조로는 불가능합니다.

1회 호출로 처리 불가능한 작업 예시

- 웹에서 최신 정보를 검색한 후 요약
- 데이터베이스를 조회하여 보고서 생성
- 파일을 읽고 분석 후 다른 파일에 저장
- 여러 단계의 조건 분기가 있는 워크플로우
- 사용자와 다중 턴 대화 후 최종 판단

루프가 필요한 이유

모델은 "도구를 호출하겠다"는 의도만 반환합니다. 실제 도구 실행, 결과를 다시 모델에 입력, 다음 판단까지의 과정이 반복 루프를 필요로 합니다. 이 루프를 매번 개발자가 구현하는 것은 낭비이며, 일관성도 떨어집니다.

```
# 루프 없이 도구 연동 시도 (문제 있는 방식)
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "오늘 서울 날씨는?"}],
    tools=[weather_tool_schema]
)
# 모델은 tool_call을 반환했지만 실행은 개발자 몫
# → 실행 후 결과를 다시 넣는 루프를 직접 구현해야 함
```

실무 포인트: Agents SDK는 이 루프를 Runner.run()이 소유합니다. 개발자는 루프 로직을 작성하는 대신 Agent 정의와 Tool 정의에만 집중할 수 있습니다.

A. 큰 그림 이해

왜 상태, 도구, 승인, 추적이 필요한가

에이전트가 단순 챗봇과 다른 것은 "행동"과 "기억"을 갖는다는 점입니다. 이 두 가지를 안전하게 다루기 위해 네 가지 기능이 반드시 필요합니다.

상태(STATE)

여러 턴에 걸친 대화에서 이전 맥락을 유지해야 합니다. 상태 없이는 에이전트가 매 호출마다 기억을 잃습니다. 세션과 RunState가 이를 해결합니다.

도구(TOOLS)

LLM은 텍스트만 생성합니다. 실제 작업(검색, 계산, DB 접근)은 외부 도구를 통해 수행됩니다. SDK는 도구 등록과 실행을 구조화합니다.

승인(APPROVAL)

자동화된 에이전트가 돌이킬 수 없는 행동(파일 삭제, 결제 등)을 취하기 전에 사람이 검토해야 합니다. HITL과 Guardrail이 이 역할을 합니다.

추적(TRACING)

에이전트가 왜 그런 결과를 냈는지, 어느 단계에서 실패했는지 파악하려면 실행 흐름 전체가 기록되어야 합니다. 로그만으로는 충분하지 않습니다.

A. 큰 그림 이해

SDK가 해결하는 범위와 해결하지 않는 범위

Agents SDK는 강력하지만 모든 것을 해결해주지는 않습니다. SDK의 범위를 명확히 알아야 설계 시 잘못된 기대를 갖지 않을 수 있습니다.

✅ SDK가 처리하는 것	❌ SDK가 처리하지 않는 것
에이전트 실행 루프 (모델 호출 반복)	비즈니스 로직 설계 (무엇을 할지)
도구 call 파싱 및 함수 실행	도구 함수 내부 구현
핸드오프 메커니즘 (에이전트 간 전환)	어떤 에이전트에 어떤 역할을 줄지
트레이싱 데이터 생성	트레이싱 분석 및 모니터링 인프라
가드레일 실행 시점 관리	가드레일 규칙 정의
세션 저장소 인터페이스	실제 데이터베이스 운영
스트리밍 이벤트 발행	프론트엔드 UI 구현

한 줄 정리: SDK는 "어떻게 실행할 것인가"를 담당합니다. "무엇을 실행할 것인가"는 여전히 개발자와 기획자가 설계해야 합니다.

A. 큰 그림 이해

챗봇 프레임워크와의 차이: LANGCHAIN, LLAMAINDEX와 비교

Agents SDK를 처음 접하는 개발자들이 자주 묻는 질문 중 하나가 "기존에 쓰던 LangChain이나 LlamaIndex와 무엇이 다른가?"입니다. 포지셔닝이 다릅니다.

구분	LangChain	LlamaIndex	OpenAI Agents SDK
주요 목적	LLM 앱 체인 구성	RAG 및 데이터 연결	에이전트 런타임 오케스트레이션
모델 지원	멀티 벤더	멀티 벤더	OpenAI 중심 (확장 가능)
추상화 수준	체인/파이프라인	인덱스/쿼리엔진	에이전트 루프/런타임
도구 통합	Tool 래퍼 방식	Tool 래퍼 방식	네이티브 함수 시그니처 방식
Tracing	LangSmith 별도	별도 설정 필요	SDK 내장
OpenAI 기능 활용	간접 지원	간접 지원	Responses API 직접 활용

Agents SDK는 OpenAI 생태계에 최적화된 에이전트 전용 런타임입니다. 멀티 벤더 환경이 필요하다면 LangChain이 더 유연할 수 있습니다. 그러나 OpenAI 모델을 중심으로 프로덕션 에이전트를 구축한다면 Agents SDK가 가장 직관적이고 최신 기능을 빠르게 활용할 수 있습니다.

핵심 아키텍처

AGENT, RUNNER, 그리고 실행 루프

Agent 정의 · Runner 실행 · 루프 단계 · 결과 객체 이해

B. 핵심 아키텍처

AGENT란 무엇인가: 정의와 속성

`Agent`는 Agents SDK에서 행동의 주체입니다. 이름, 지시사항, 모델, 도구, 핸드오프 목록을 가지며, "이 에이전트는 무엇을 할 수 있고 어떻게 행동해야 하는가"를 선언합니다. Agent 자체는 실행 객체가 아니라 **설정 객체**입니다.

```
from agents import Agent

agent = Agent(
    name="고객지원 에이전트",
    instructions="당신은 친절한 고객지원 담당자입니다. 한국어로 답변합니다.",
    model="gpt-4o",
    tools=[search_kb, create_ticket],
    output_type=SupportResponse,  # 구조화된 출력 타입
)
```

속성	설명
`name`	에이전트 식별자. 트레이싱과 핸드오프에서 사용됨.
`instructions`	시스템 프롬프트. 문자열 또는 컨텍스트를 받는 함수로 동적 설정 가능.
`model`	사용할 모델명. 기본값은 환경변수 또는 SDK 기본값.
`model_settings`	temperature, max_tokens 등 모델 파라미터.
`tools`	이 에이전트가 호출할 수 있는 도구 목록.
`handoffs`	이 에이전트가 위임할 수 있는 다른 에이전트 목록.
`output_type`	최종 출력의 Pydantic 모델 타입. 구조화된 출력 강제.
`guardrails`	이 에이전트에 적용할 가드레일 목록.
`hooks`	실행 생명주기(시작, 종료, 도구 호출 전후 등)에 실행할 콜백.

B. 핵심 아키텍처

RUNNER란 무엇인가: AGENT와의 역할 분리

Runner는 에이전트 루프를 실행하는 엔진입니다. Agent가 "무엇을 할 수 있는가"를 정의한다면, Runner는 "어떻게 실행할 것인가"를 담당합니다. 이 역할 분리는 설계상 매우 중요한 결정입니다.

AGENT의 역할

- 이름, 지시사항 선언
- 사용 가능한 도구 목록
- 핸드오프 가능 에이전트 목록
- 출력 타입 및 가드레일
- 실행과 무관한 순수 설정

RUNNER의 역할

- 입력을 받아 루프 시작
- 모델 호출 및 응답 파싱
- tool call 감지 및 함수 실행
- handoff 감지 및 에이전트 전환
- final output 판정 후 루프 종료
- RunResult 반환

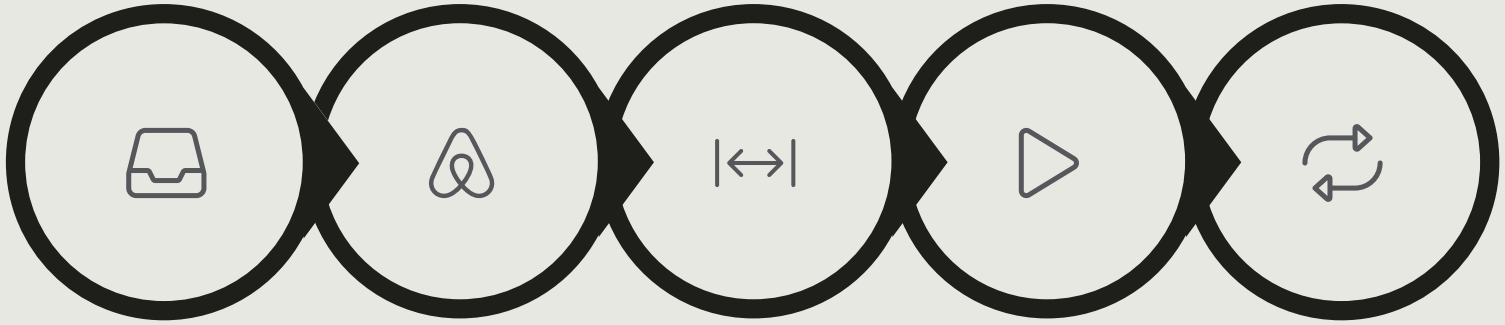
실행 메서드	특징	적합한 사용처
Runner.run()	비동기(async), 전체 완료 후 결과 반환	FastAPI, async 환경
Runner.run_sync()	동기, 내부에서 이벤트 루프 실행	스크립트, Jupyter, 동기 환경
Runner.run_streamed()	비동기, 이벤트 스트림 반환	실시간 UI, 챗 인터페이스

실무 포인트: Agent 객체를 재사용하면서 Runner만 다른 입력으로 호출하는 패턴이 일반적입니다. Agent는 인스턴스화 비용이 낮으므로 필요하다면 매 요청마다 새로 생성해도 됩니다.

B. 핵심 아키텍처

에이전트 루프 상세: 한 스텝씩 이해하기

Runner가 실행하는 에이전트 루프는 다음 단계를 반복합니다. 이 루프를 이해하면 에이전트가 "왜 그런 행동을 했는지" 디버깅하기 훨씬 쉬워집니다.



루프 종료 조건

- **final output:** 모델이 도구 호출 없이 텍스트 또는 구조화된 출력을 반환
- **max_turns 초과:** 최대 반복 횟수 초과 시 예외 발생
- **guardrail tripwire:** 가드레일이 실행을 차단
- **pause (HITL):** 사람 승인 대기 상태로 전환

루프 디버깅 포인트

- 루프가 무한히 도는 경우 → max_turns 설정 확인
- 도구가 실행되지 않는 경우 → 도구 스키마와 instructions 점검
- 항상 첫 응답에서 종료되는 경우 → output_type 또는 지시사항 점검

B. 핵심 아키텍처

실행 결과 객체 이해: RUNRESULT와 주요 속성

`Runner.run()`이 반환하는 `RunResult` 객체는 에이전트 실행의 전체 결과를 담고 있습니다. 이 객체를 제대로 이해해야 결과를 올바르게 활용할 수 있습니다.

```
result = await Runner.run(agent, input="서울 날씨 알려줘")

# 최종 텍스트 출력
print(result.final_output)

# 실행 중 발생한 모든 새 아이템 (tool calls, messages 등)
print(result.new_items)

# 마지막으로 실행된 에이전트
print(result.last_agent)

# 다음 대화에 사용할 수 있는 입력 형태로 변환
next_input = result.to_input_list()
```

속성	의미
`final_output`	에이전트가 최종적으로 반환한 텍스트 또는 구조화된 객체
`new_items`	이번 실행에서 생성된 모든 아이템 목록 (메시지, 도구 호출, 핸드오프 등)
`last_agent`	루프가 끝날 때 제어권을 갖고 있던 에이전트
`raw_responses`	모델이 반환한 원시 API 응답 목록
`to_input_list()`	다음 `Runner.run()` 호출 시 대화 이력으로 사용 가능한 형태로 변환

한 줄 정리: result.final_output은 최종 답변, result.to_input_list()는 대화를 이어가기 위한 이력입니다. 이 두 가지만 알아도 기본 흐름을 구현할 수 있습니다.

C. 모델 계층

모델 계층

SDK가 모델 API 위의 런타임인 이유

Responses API · Chat Completions · Provider 추상화 · 모델 선택 기준

C. 모델 계층

SDK의 모델 추상화: 두 가지 경로

Agents SDK는 모델 호출을 추상화하여 개발자가 모델 API의 세부 사항을 직접 다루지 않아도 되도록 설계되어 있습니다. 두 가지 주요 경로가 있습니다.

RESPONSES API 경로 (기본값)

OpenAI의 최신 API 경로. Built-in 도구(웹 검색, 파일 검색, 코드 인터프리터)를 네이티브로 지원합니다. `previous_response_id`를 통한 서버 측 대화 상태 관리도 지원합니다.

```
agent = Agent(
    name="researcher",
    model="gpt-4o",  # Responses API 사용
    tools=[WebSearchTool()],
)
```

CHAT COMPLETIONS 경로

기존 Chat Completions API를 사용하는 경로. OpenAI 모델 외에 OpenAI 호환 API를 제공하는 다른 벤더도 사용 가능합니다.

```
from agents.extensions.models.litellm_model import LitellmModel

agent = Agent(
    name="researcher",
    model=LitellmModel(model="anthropic/claude-3-5-sonnet"),
)
```

모델 선택이 중요한 경우 VS 그렇지 않은 경우

모델 선택이 결정적으로 중요한 경우	모델 선택이 상대적으로 덜 중요한 경우
복잡한 추론, 코드 생성, 다단계 계획 수립	단순 분류, 키워드 추출, 형식 변환
도구 호출이 정확해야 하는 프로덕션 에이전트	내부 프로토타입, PoC 단계
비용이 크게 중요한 고빈도 호출	저빈도 내부 업무 도구

실무 포인트: 에이전트 설계 초기에는 모델 선택보다 도구 설계, 지시사항 작성, 루프 구조에 집중하는 것이 더 효과적입니다. 모델은 나중에 교체할 수 있지만 아키텍처는 바꾸기 어렵습니다.

도구 계층

TOOLS: 에이전트의 손과 발

Function Tools · Hosted Tools · 도구 설계 패턴 · tool_use_behavior

D. 도구 계층

PYTHON FUNCTION TOOL: 함수 시그니처에서 스키마까지

Agents SDK에서 가장 기본적인 도구 유형은 Python 함수입니다. SDK는 함수의 시그니처, 타입 힌트, docstring을 분석하여 모델이 이해하는 JSON 스키마를 자동으로 생성합니다. 개발자는 별도 스키마 작성 없이 일반 Python 함수를 도구로 사용할 수 있습니다.

```python
from agents import function_tool

@function_tool
def get_weather(city: str, unit: str = "celsius") -> str:
    """지정된 도시의 현재 날씨를 반환합니다.

    Args:
        city: 날씨를 조회할 도시 이름
        unit: 온도 단위 (celsius 또는 fahrenheit)
    """
    # 실제 날씨 API 호출
    return f"{city}의 현재 온도는 22도입니다."

agent = Agent(
    name="weather-agent",
    tools=[get_weather],  # 함수를 그대로 등록
)
```

자동 스키마 생성 규칙

- 함수명 → tool name
- docstring 첫 줄 → tool description
- 파라미터 타입 힌트 → JSON schema 타입
- 기본값 없는 파라미터 → required 필드
- Args 섹션 → 각 파라미터 설명

좋은 도구 함수 설계 원칙

- 함수명은 동사+명사 형태로 명확하게
- docstring은 반드시 작성 (모델이 읽음)
- 타입 힌트는 항상 명시
- 파라미터는 5개 이하로 제한
- 반환값은 문자열 또는 직렬화 가능한 타입

D. 도구 계층

HOSTED TOOLS: OPENAI가 제공하는 내장 도구

Hosted Tools는 OpenAI 서버에서 실행되는 도구들로, 개발자가 별도 구현 없이 즉시 사용할 수 있습니다. Responses API 경로에서만 사용 가능합니다.

🔍 WEBSEARCHTOOL

실시간 웹 검색. 최신 정보가 필요한 에이전트에 적합. 검색 쿼리를 자동으로 구성하고 결과를 요약하여 반환.

```
from agents import WebSearchTool
tools=[WebSearchTool()]
```

📁 FILESEARCHTOOL

Vector Store 기반 파일 검색. 업로드된 문서에서 관련 내용을 시맨틱 검색으로 찾아 반환. RAG 패턴 구현에 적합.

```
from agents import FileSearchTool
tools=[FileSearchTool(vector_store_ids=["vs_xxx"])]
```

💻 CODEINTERPRETERTOOL

Python 코드 실행 환경. 데이터 분석, 계산, 차트 생성 등 코드 실행이 필요한 작업에 사용. 샌드박스 환경에서 실행됨.

```
from agents import CodeInterpreterTool
tools=[CodeInterpreterTool()]
```

🖼️ IMAGEGENERATIONTOOL

DALL-E를 통한 이미지 생성. 텍스트 설명으로 이미지를 생성하고 URL을 반환. 창작, 마케팅 에이전트에 활용.

```
from agents import ImageGenerationTool
tools=[ImageGenerationTool()]
```

D. 도구 계층

TOOL_USE_BEHAVIOR: 도구 실행 후 흐름 제어

tool_use_behavior는 도구가 실행된 이후 에이전트 루프가 어떻게 동작할지를 제어합니다. 기본 동작은 도구 결과를 모델에 다시 전달하여 최종 응답을 생성하는 것이지만, 경우에 따라 즉시 종료가 더 적합합니다.

설정값	동작	적합한 상황
run_llm_again (기본값)	도구 결과를 모델에 전달 → 모델이 최종 응답 생성	도구 결과를 해석하거나 요약해야 할 때
stop_on_first_tool	첫 번째 도구 실행 결과를 즉시 final output으로 사용	도구가 이미 완성된 결과를 반환할 때
커스텀 함수	도구 호출 목록을 받아 계속/중단 여부를 동적으로 결정	특정 도구만 즉시 종료, 나머지는 계속 처리

```
# 도구 결과를 그대로 반환 (모델 재호출 없음)
agent = Agent(
    name="lookup-agent",
    tools=[search_database],
    tool_use_behavior="stop_on_first_tool",  # 검색 결과 즉시 반환
)

# 특정 도구만 즉시 종료
agent = Agent(
    name="smart-agent",
    tools=[quick_lookup, complex_analysis],
    tool_use_behavior=StopAtTools(stop_at_tool_names=["quick_lookup"]),
)
```

실무 포인트: 도구가 이미 잘 포맷된 응답을 반환한다면 stop_on_first_tool로 불필요한 모델 호출을 줄일 수 있습니다. 비용과 응답 속도 모두 개선됩니다.

D. 도구 계층

도구 설계: 좋은 패턴과 피해야 할 패턴

도구의 품질이 에이전트의 품질을 결정합니다. 잘못 설계된 도구는 모델이 잘못 호출하게 만들고, 예측 불가능한 동작을 유발합니다.

✅ 좋은 도구 설계 패턴

- **명확한 목적:** 도구 하나는 하나의 역할만. 다목적 도구 지양
- **설명 충분:** docstring에 언제 써야 하는지, 반환값이 무엇인지 명시
- **예측 가능한 반환:** 항상 같은 구조로 결과 반환 (dict, str)
- **오류 처리:** 예외 발생 시 의미 있는 오류 메시지 반환
- **멱등성 고려:** 반복 호출해도 부작용 없는 도구 우선 설계

❌ 피해야 할 도구 설계 패턴

- **모호한 이름:** `do_something()`, `process()` 등 추상적 이름
- **docstring 없음:** 모델이 언제 쓸지 판단 불가
- **너무 많은 파라미터:** 모델이 잘못 채울 가능성 증가
- **예외를 raise:** 에이전트 루프 중단 위험. 오류 문자열 반환 권장
- **부작용 숨기기:** 데이터 수정, 이메일 발송 등을 이름에서 숨기면 위험

⚠ **실무 포인트:** "도구가 뭘 하는지 모델이 이름과 docstring만 보고 정확히 판단할 수 있는가?"를 항상 질문하세요. 모델은 코드 내부를 보지 못합니다. docstring이 전부입니다.

멀티에이전트

HANDOFFS와 오케스트레이션 패턴

Handoff · Triage Agent · Manager 패턴 · 복잡성 경고

E. 멀티에이전트와 오케스트레이션

HANDOFF란 무엇인가: TOOL과의 차이

Handoff는 한 에이전트가 다른 에이전트에게 대화의 제어권을 완전히 넘기는 메커니즘입니다. 이는 도구 호출과 근본적으로 다릅니다.

구분	Tool 호출	Handoff
제어권	현재 에이전트 유지. 도구 결과 받아서 계속	대상 에이전트로 완전 이전. 현재 에이전트 종료
대화 이력	현재 에이전트의 이력 유지	대상 에이전트가 이력 인수
적합한 상황	현재 에이전트가 판단을 유지해야 할 때	전문 에이전트가 처음부터 끝까지 처리해야 할 때
사용 예시	날씨 조회, DB 검색, 계산	기술 지원 → 엔지니어 에이전트, 한국어 → 번역 에이전트

```
from agents import Agent, handoff

billing_agent = Agent(name="결제 전문 에이전트", instructions="결제 관련 문의를 처리합니다.")
tech_agent = Agent(name="기술 지원 에이전트", instructions="기술적 문제를 해결합니다.")

triage_agent = Agent(
    name="접수 에이전트",
    instructions="문의 유형에 따라 적절한 전문 에이전트에게 연결합니다.",
    handoffs=[billing_agent, tech_agent],  # 핸드오프 가능 에이전트 목록
)
```

실무 포인트: "이 에이전트가 결과를 받아서 판단을 계속해야 하는가?" → Tool. "이 에이전트의 역할이 끝나고 전문가가 처음부터 처리해야 하는가?" → Handoff.

E. 멀티에이전트와 오케스트레이션

오케스트레이션 패턴 2가지: TRIAGE와 MANAGER

멀티에이전트 시스템에서 가장 일반적으로 사용되는 두 가지 오케스트레이션 패턴을 비교합니다.

TRIAGE AGENT 패턴

입구 에이전트가 요청을 분류하여 적합한 전문 에이전트로 핸드오프합니다. 전문 에이전트가 처음부터 끝까지 처리합니다.

- 구조: 1 Triage → N 전문 에이전트
- 흐름: 분류 → 이전 → 전문 처리 → 완료
- 적합: 고객 지원, 문의 라우팅
- 장점: 단순, 명확한 책임 분리

MANAGER + AGENTS AS TOOLS 패턴

오케스트레이터 에이전트가 하위 에이전트들을 도구처럼 호출하고 결과를 통합합니다. 제어권은 매니저가 유지합니다.

- 구조: 1 Manager → N 하위 에이전트(도구처럼)
- 흐름: 계획 → 하위 에이전트 호출 → 결과 통합 → 완료
- 적합: 복잡한 리서치, 다단계 분석
- 장점: 유연한 조합, 결과 통합 가능

⚠ **경고:** 멀티에이전트는 강력하지만 복잡성도 함께 증가합니다. 처음부터 멀티에이전트를 설계하는 것보다, 단일 에이전트로 시작하여 필요할 때 분리하는 접근이 훨씬 안전합니다.

E. 멀티에이전트와 오케스트레이션

과도한 에이전트 분할이 왜 위험한가

멀티에이전트의 가장 흔한 실수는 "기능마다 에이전트를 만드는 것"입니다. 이는 마이크로서비스의 과도한 분리와 유사한 문제를 일으킵니다.

문제 1: 컨텍스트 손실

핸드오프가 일어날 때마다 이전 에이전트의 판단 맥락이 새 에이전트에 완전히 전달되지 않을 수 있습니다. 복잡한 멀티에이전트 체인은 "전화 돌리기 게임"처럼 맥락이 희석됩니다.

문제 2: 디버깅 난이도 급증

단일 에이전트의 실패는 한 곳을 보면 됩니다. 6개 에이전트가 연결된 시스템에서 실패가 발생하면 어느 에이전트, 어느 단계, 어떤 핸드오프에서 문제가 생겼는지 추적이 어렵습니다.

문제 3: 비용과 지연 증가

에이전트 하나를 거칠 때마다 모델 호출이 발생합니다. 4개 에이전트를 순서대로 거친다면 최소 4회의 모델 호출이 필요합니다. 불필요한 분리는 비용과 응답 속도에 직접적인 영향을 줍니다.

문제 4: 책임 경계 모호성

에이전트를 너무 잘게 나누면 어떤 에이전트가 어떤 결정을 내려야 하는지 불분명해집니다. 에이전트들이 서로 다른 에이전트로 계속 핸드오프하는 "핑퐁" 상황이 발생하기도 합니다.

한 줄 정리: 에이전트 분리의 기준은 "역할의 차이"가 아니라 "전문성과 컨텍스트의 단절"이어야 합니다.

F. 상태, 메모리, 세션

상태 · 메모리 · 세션

대화 상태를 구조화하는 방법

---

to_input_list · Session 저장소 · 단기 기억 vs 컨텍스트 분리

F. 상태, 메모리, 세션

대화 상태를 직접 다뤄야 하는 이유

LLM은 기본적으로 stateless입니다. 각 모델 호출은 독립적이며, 이전 대화를 기억하지 않습니다. 멀티 턴 대화를 구현하려면 개발자가 상태를 명시적으로 관리해야 합니다.

방법 1: TO_INPUT_LIST 방식

가장 직접적인 방식. result.to_input_list()로 이전 실행 결과를 다음 호출의 입력으로 변환합니다. 상태를 코드에서 직접 관리합니다.

```
# 첫 번째 턴
result = await Runner.run(agent, "안녕하세요")
history = result.to_input_list()

# 두 번째 턴 (이전 대화 포함)
result = await Runner.run(
    agent,
    history + [{"role": "user", "content": "제 이름을 기억하나요?"}]
)
```

방법 2: SESSION 기반 방식

Session 객체를 사용하면 상태 관리를 SDK에 위임할 수 있습니다. 저장소(SQLite, Redis 등)와 연결하면 대화 이력이 자동으로 저장/로드됩니다.

```
from agents.sessions import SQLiteSession

session = SQLiteSession(
    session_id="user_123",
    db_path="conversations.db"
)

result = await Runner.run(
    agent,
    "안녕하세요",
    session=session  # 세션 자동 관리
)
```

구분	to_input_list 방식	Session 방식
복잡도	낮음 (직접 관리)	낮음 (SDK 위임)
영속성	코드에서 직접 저장 필요	저장소에 자동 저장
적합한 상황	단순 스크립트, 단일 대화	웹 앱, 멀티 유저 서비스

F. 상태, 메모리, 세션

세션 저장소 선택 기준과 단기/장기 기억 분리

세션 저장소 선택은 서비스 규모와 요구사항에 따라 달라집니다. 또한 "세션에 무엇을 저장할 것인가"도 중요한 설계 결정입니다.

저장소	특징	적합한 상황	주의점
메모리 (기본값)	프로세스 생명주기 동안만 유지	테스트, 단일 세션 스크립트	서버 재시작 시 손실
SQLite	로컬 파일 기반, 설정 간단	소규모 앱, 개인 도구	동시성 한계
Redis	인메모리, 고속, TTL 지원	고빈도 요청, 캐시 필요시	서버 비용, 데이터 손실 가능
SQLAlchemy (RDB)	관계형 DB, 강력한 쿼리	기업 서비스, 감사 추적 필요시	스키마 설계 필요

단기 기억 (세션 컨텍스트)

현재 대화의 이력. 모델에 직접 전달됩니다. 너무 길어지면 토큰 비용 증가 및 성능 저하가 발생합니다. 일반적으로 최근 N턴 또는 요약본을 유지합니다.

서버 컨텍스트 (RUNCONTEXTWRAPPER)

사용자 정보, 권한, 설정 등 코드 실행에 필요한 데이터. 모델에 전달되지 않고 도구 함수에서만 접근합니다. 세션과 별개로 관리됩니다.

한 줄 정리: 모델이 읽어야 하는 것은 세션(단기 기억), 코드가 읽어야 하는 것은 컨텍스트(서버 상태)로 분리하세요.

컨텍스트와 의존성 주입

RUNCONTEXTWRAPPER로 코드 런타임 컨텍스트 다루기

LLM 컨텍스트 vs 코드 컨텍스트 · 사용자 정보 · 권한 주입

G. 컨텍스트와 의존성 주입

RUNCONTEXTWRAPPER: LLM과 코드 컨텍스트의 분리

에이전트를 실행할 때 두 종류의 컨텍스트가 필요합니다. 이 둘을 구분하지 않으면 보안 문제나 과도한 토큰 사용으로 이어집니다.

LLM 컨텍스트 (프롬프트)

모델이 읽는 정보. instructions, 대화 이력, 도구 결과가 여기에 해당합니다. 토큰으로 소비되므로 최소한으로 유지해야 합니다.

- 시스템 프롬프트 (instructions)
- 대화 이력 (session)
- 도구 실행 결과

코드 런타임 컨텍스트 (RUNCONTEXTWRAPPER)

도구 함수가 접근하는 서버 측 데이터. 모델에게는 보이지 않으며, Python 코드에서만 접근 가능합니다.

- 인증된 사용자 ID, 권한 정보
- 로거, 데이터베이스 연결
- 외부 API 클라이언트
- 요청별 설정값

```python
from dataclasses import dataclass
from agents import RunContextWrapper, function_tool

@dataclass
class AppContext:
    user_id: str
    db_client: DatabaseClient
    permissions: list[str]

@function_tool
async def get_user_orders(ctx: RunContextWrapper[AppContext]) -> str:
    """현재 사용자의 주문 목록을 조회합니다."""
    user_id = ctx.context.user_id  # LLM에 노출되지 않는 컨텍스트
    return await ctx.context.db_client.get_orders(user_id)

result = await Runner.run(
    agent,
    "내 주문 목록 보여줘",
    context=AppContext(user_id="u_123", db_client=db, permissions=["read"])
)
```

⚠ **주의:** 민감한 정보(사용자 ID, 권한, API 키)를 instructions나 대화 이력에 넣으면 모델 응답에 노출될 수 있습니다. 반드시 RunContextWrapper를 통해 코드 레벨에서만 접근하세요.

GUARDRAILS & HITL

통제 가능한 자동화를 위한 안전 계층

입력/출력 가드레일 · Tripwire · Human-in-the-Loop · Pause/Resume

H. GUARDRAILS / HITL / 통제

GUARDRAILS: 세 가지 유형과 동작 방식

Guardrails는 에이전트 실행의 안전 장치입니다. 입력, 출력, 도구 호출 단계에서 검증을 수행하며, 위반 시 실행을 차단하거나 경고를 발생시킵니다.

INPUT GUARDRAIL

에이전트 실행 전, 사용자 입력을 검증합니다. 욕설, 금지 주제, 불법 요청 등을 차단합니다. 병렬로 실행되어 메인 에이전트와 동시에 처리됩니다.

OUTPUT GUARDRAIL

에이전트 응답 생성 후, 최종 출력을 검증합니다. 민감 정보 노출, 브랜드 가이드라인 위반, 부적절한 내용 등을 필터링합니다.

TOOL GUARDRAIL

도구 호출 전후에 검증합니다. 특정 도구의 파라미터 검증, 호출 빈도 제한, 권한 확인 등에 활용합니다.

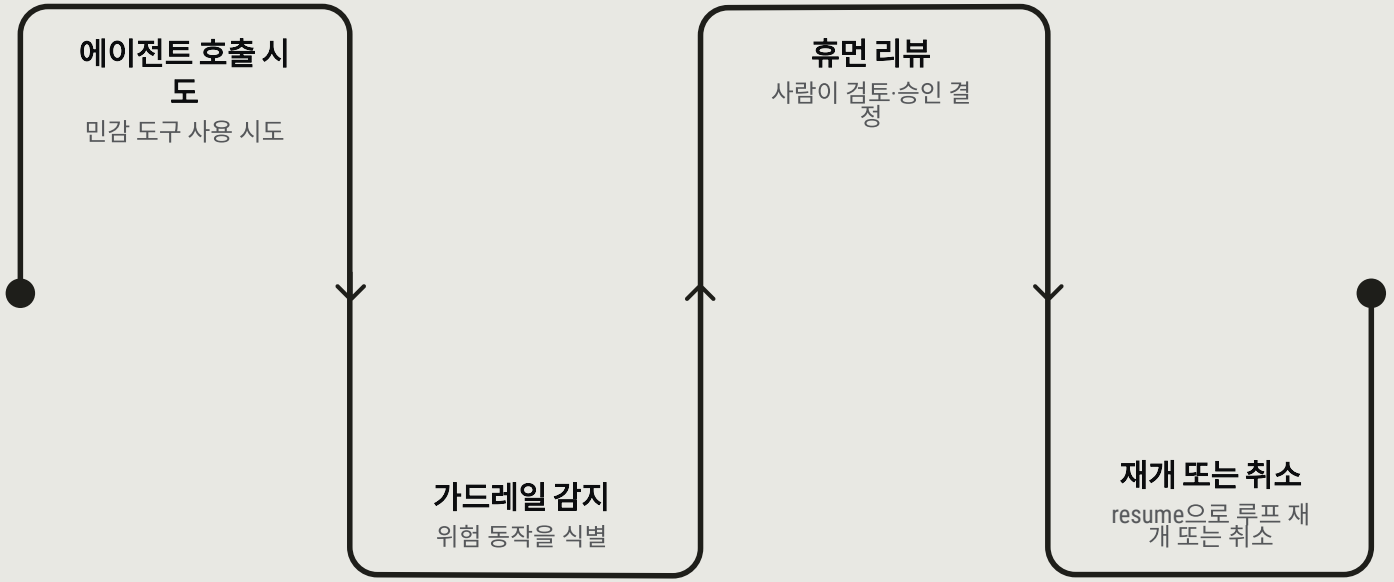
```python
from agents import input_guardrail, GuardrailFunctionOutput

@input_guardrail
async def check_topic(ctx, agent, input) -> GuardrailFunctionOutput:
    """금지 주제 감지 가드레일"""
    is_blocked = await detect_forbidden_topic(input)
    return GuardrailFunctionOutput(
        output_info={"topic": "blocked"} if is_blocked else {},
        tripwire_triggered=is_blocked,  # True이면 실행 차단
    )

agent = Agent(
    name="safe-agent",
    input_guardrails=[check_topic],  # 가드레일 등록
)
```

병렬 실행의 의미: Input guardrail은 메인 에이전트 실행과 병렬로 동작합니다. 가드레일이 위반을 감지하면 tripwire를 발동하여 메인 실행을 중단시킵니다. 레이턴시 오버헤드가 최소화됩니다.

H. GUARDRAILS / HITL / 통제

HUMAN-IN-THE-LOOP: 사람이 개입하는 승인 흐름

에이전트가 돌이킬 수 없는 행동(이메일 발송, 결제, 데이터 삭제 등)을 취하기 전에 사람이 검토하고 승인하는 흐름입니다. RunState의 pause/resume 메커니즘을 통해 구현합니다.




```python
# 승인이 필요한 도구
@function_tool
async def send_email(ctx: RunContextWrapper, to: str, body: str) -> str:
    """이메일을 발송합니다. 반드시 사람 승인 후 실행됩니다."""
    # 이 함수는 승인 후에만 실행됨
    return await email_client.send(to=to, body=body)

# 승인 가드레일
@tool_guardrail
async def require_approval(ctx, tool_call) -> GuardrailFunctionOutput:
    if tool_call.tool_name == "send_email":
        # pause 상태로 전환 → 사람 검토 필요
        return GuardrailFunctionOutput(tripwire_triggered=True)
    return GuardrailFunctionOutput(tripwire_triggered=False)
```

⊗ **실무 포인트:** HITL은 비용이 높습니다. 모든 도구에 적용하면 자동화의 의미가 없어집니다. "돌이킬 수 없는 행동", "높은 비용 발생", "외부 발신"에만 선택적으로 적용하세요.

I. 스트리밍 / 실시간 / 음성

스트리밍 & 실시간

RUN_STREAMED와 VOICE PIPELINE

스트리밍 이벤트 · 실시간 에이전트 · 텍스트 vs 음성 선택 기준

I. 스트리밍 / 실시간 / 음성

RUN_STREAMED: 이벤트 기반 스트리밍 이해

사용자 경험 관점에서 전체 응답이 완성될 때까지 기다리는 것보다 생성되는 대로 보여주는 것이 훨씬 자연스럽습니다. `run_streamed()`는 이를 위한 API입니다.

```
result = Runner.run_streamed(agent, "AI 트렌드 요약해줘")

async for event in result.stream_events():
    if event.type == "raw_response_event":
        # 토큰 단위 스트리밍 (ChatGPT처럼)
        print(event.data.delta, end="", flush=True)
    elif event.type == "run_item_stream_event":
        # 완성된 아이템 (tool call, message 등)
        if event.item.type == "tool_call_item":
            print(f"\n[도구 호출: {event.item.name}]")
    elif event.type == "agent_updated_stream_event":
        # 에이전트 전환 이벤트 (handoff 발생 시)
        print(f"\n[에이전트 전환: {event.new_agent.name}]")
```

이벤트 타입	의미
`raw_response_event`	모델의 토큰 단위 스트리밍 데이터
`run_item_stream_event`	완성된 아이템 (메시지, 도구 호출, 핸드오프 등)
`agent_updated_stream_event`	현재 에이전트가 변경됨 (핸드오프 발생)

실무 포인트: 스트리밍 UI를 구현할 때는 raw_response_event로 텍스트를 점진적으로 보여주고, run_item_stream_event로 도구 호출 상태를 사용자에게 알려주는 패턴이 좋습니다.

I. 스트리밍 / 실시간 / 음성

VOICE PIPELINE: 음성 에이전트 개요와 선택 기준

Agents SDK는 텍스트 에이전트 외에도 Voice Pipeline을 통한 음성 기반 에이전트를 지원합니다. 두 방식은 용도와 설계가 다릅니다.

구분	텍스트 에이전트	실시간/음성 에이전트
입출력	텍스트 입력, 텍스트 출력	오디오 입력, 오디오 출력
레이턴시	수 초 허용 가능	수백ms 이내 목표
상태 관리	세션, to_input_list	실시간 스트림 기반
사용 도구	모든 도구 유형	빠른 실행 도구 권장
적합한 제품	챗봇, 내부 도구, 자동화	음성 어시스턴트, 콜센터 봇

VOICE PIPELINE 구성

- STT (음성 → 텍스트): Whisper 모델
- 에이전트 실행: 일반 텍스트 에이전트
- TTS (텍스트 → 음성): TTS 모델
- 전체를 `VoicePipeline`이 연결

실시간 에이전트 (REALTIME)

- GPT-4o Realtime API 기반
- 오디오를 직접 처리 (STT/TTS 분리 없음)
- 더 낮은 레이턴시 가능
- 현재 기능 제한 있음 (도구 등)

J. MCP와 외부 시스템 연결

MCP

MODEL CONTEXT PROTOCOL로 외부 시스템 연결하기

MCP 개념 · 연결 방식 · 사내 시스템 연동 · MCP의 한계

J. MCP와 외부 시스템 연결

MCP란 무엇인가: 도구 확장 포트로 이해하기

MCP(Model Context Protocol)는 Anthropic이 제안한 표준 프로토콜로, LLM 애플리케이션과 외부 시스템 사이의 표준화된 인터페이스를 정의합니다. Agents SDK는 MCP를 도구 제공자(서버)로 연결하는 기능을 지원합니다.

MCP 없이 도구 연동 시

각 외부 시스템(Slack, GitHub, DB 등)마다 개별 Python function tool을 직접 구현해야 합니다. 도구가 많아질수록 유지보수 부담이 증가합니다.

MCP로 도구 연동 시

MCP 서버가 도구 목록과 실행 인터페이스를 표준화하여 제공합니다. Agents SDK는 MCP 서버에 연결하여 도구를 자동으로 가져옵니다. 커뮤니티가 만든 MCP 서버를 그대로 활용 가능합니다.

연결 방식	특징	적합한 상황
stdio	로컬 프로세스로 MCP 서버 실행	로컬 개발, 단일 서버 환경
HTTP/SSE	원격 HTTP 서버로 연결	중앙화된 MCP 서버, 멀티 에이전트

⚠ **MCP가 만능은 아닙니다:** MCP 서버의 품질은 누가 만들었느냐에 따라 크게 다릅니다. 프로덕션에서 외부 MCP 서버를 사용할 때는 도구의 동작, 보안, 안정성을 반드시 직접 검증하세요. 맹목적으로 신뢰하지 말 것.

TRACING & 운영

에이전트 시스템의 가시성 확보

Trace · Span · 운영 관측 · 디버깅 · 민감정보 주의

TRACING: TRACE와 SPAN의 구조

에이전트 시스템은 여러 모델 호출, 도구 실행, 핸드오프가 연결된 복잡한 실행 그래프를 만듭니다. Tracing은 이 그래프 전체를 구조화된 방식으로 기록합니다.

TRACE

하나의 에이전트 실행 전체를 의미합니다. `Runner.run()` 한 번 호출이 하나의 Trace를 생성합니다. Trace는 실행의 루트 노드입니다.

SPAN

Trace 내의 개별 작업 단위입니다. 모델 호출 하나, 도구 실행 하나, 핸드오프 하나가 각각 하나의 Span이 됩니다. Span은 계층 구조로 중첩될 수 있습니다.

Span 타입	기록하는 정보
Agent Span	에이전트 이름, 실행 시작/종료 시간, 입출력
LLM Span	모델명, 프롬프트, 응답, 토큰 사용량, 레이턴시
Tool Span	도구 이름, 입력 파라미터, 반환값, 실행 시간
Handoff Span	출발 에이전트, 도착 에이전트, 핸드오프 이유
Guardrail Span	가드레일 이름, 검증 결과, tripwire 발동 여부

ⓘ Tracing 데이터는 기본적으로 OpenAI 대시보드로 전송됩니다. `set_tracing_disabled()`로 비활성화하거나 커스텀 프로세서로 다른 시스템(Langfuse, Datadog 등)에 전송할 수 있습니다.

K. TRACING / 운영 / 프로덕션

왜 로그만으로는 부족한가: TRACING의 가치

일반적인 애플리케이션에서는 로그 파일만으로 충분할 때가 많습니다. 그러나 에이전트 시스템에서는 로그의 한계가 명확하게 드러납니다.

디버깅: "왜 이 결과가 나왔는가?"

에이전트가 어떤 도구를 어떤 순서로 호출했는지, 각 단계에서 모델이 무엇을 판단했는지를 시간 순서로 재현해야 합니다. 로그는 흐름을 보여주지 않지만 Trace는 전체 실행 그래프를 시각화합니다.

평가: "이 에이전트가 잘 작동하고 있는가?"

평균 도구 호출 횟수, 핸드오프 발생 빈도, 특정 도구의 실패율, 평균 응답 시간 등 운영 지표를 Span 데이터에서 추출할 수 있습니다. 이는 프롬프트 최적화와 구조 개선의 근거가 됩니다.

재현성: "같은 입력에서 같은 결과가 나오는가?"

에이전트는 비결정론적입니다. Trace를 저장해두면 특정 입력에서 어떤 실행 경로를 탔는지 나중에 재현하고 비교할 수 있습니다. 모델 버전 업그레이드 전후 비교에도 유용합니다.

⊗ **민감정보 주의:** Tracing은 프롬프트와 도구 입출력을 모두 기록합니다. 개인정보, 의료 정보, 금융 데이터가 포함된 경우 `trace_include_sensitive_data=False` 설정이나 커스텀 프로세서로 민감 필드를 마스킹해야 합니다.

L. 실전 설계 관점

실전 설계

현업에서 바로 쓸 수 있는 설계 가이드

권장 시작 패턴 · 아키텍처 예시 · 안티패턴 · 도입 체크리스트

L. 실전 설계 관점

단일 에이전트에서 시작하라: 권장 설계 패턴

많은 개발자들이 처음부터 멀티에이전트 시스템을 설계하려 합니다. 이는 대부분의 경우 과도한 복잡성을 초래합니다. OpenAI가 권장하는 접근법은 단일 에이전트에서 시작하여 필요에 따라 확장하는 것입니다.

단계	포함 요소	언제 다음 단계로?
1단계: 기본	Agent + 2~5개 Tools + run_sync	기본 기능이 검증되었을 때
2단계: 운영화	Session + Input/Output Guardrails + Tracing	복수 사용자에게 서비스할 때
3단계: 확장	Handoffs + 전문 에이전트 분리 + HITL	단일 에이전트 지시사항이 너무 길어질 때

실무 포인트: "지시사항이 2,000 토큰을 넘어가기 시작한다", "도구가 10개 이상이 되어 모델이 선택을 잘 못한다", "완전히 다른 도메인 전문성이 필요하다"가 에이전트 분리를 고려하는 신호입니다.

L. 실전 설계 관점

아키텍처 예시 4가지: 유형별 설계 가이드

현업에서 자주 만나는 4가지 유형의 에이전트 시스템과 각각의 설계 포인트를 정리합니다.

1

FAQ 봇 / 고객 응대 에이전트

구성: 단일 에이전트 + FileSearchTool + 세션
포인트: 지식베이스 최신화, Input Guardrail(욕설 차단), 답변 불가 시 에스컬레이션 핸드오프
주의: output_type으로 구조화된 응답 강제, 환각 방지

2

내부 업무 도우미 에이전트

구성: 단일 에이전트 + 사내 API 도구 + RunContextWrapper(사용자 권한)
포인트: 사용자 권한을 컨텍스트로 주입, 권한 초과 도구 호출 방지
주의: 도구에서 권한 검증 필수, 감사 로그 Tracing

3

문서 기반 분석 에이전트

구성: 단일 에이전트 + FileSearchTool + CodeInterpreterTool
포인트: 대용량 문서는 Vector Store 전처리 필요, 코드 실행 결과를 요약 후 응답
주의: 토큰 비용 관리, 코드 실행 샌드박스 보안

4

액션형 자동화 에이전트

구성: Triage + 전문 에이전트 + HITL Guardrail
포인트: 돌이킬 수 없는 액션에 반드시 HITL 적용, 각 단계 Tracing 필수
주의: 에이전트 분리 기준 명확히, 루프 최대 횟수(max_turns) 설정

L. 실전 설계 관점

실패하기 쉬운 설계 안티패턴 10가지

수많은 에이전트 프로젝트에서 반복적으로 나타나는 설계 실수를 정리했습니다. 이 중 3개 이상 해당된다면 설계를 재검토할 필요가 있습니다.

① 지시사항 과적재

지시사항에 모든 규칙을 넣으려는 시도. 길어질수록 모델이 무시하는 부분이 생깁니다.

② 도구 난립

10개 이상의 도구를 한 에이전트에 부여. 모델이 어느 도구를 써야 할지 판단을 못합니다.

③ 가드레일 부재

입출력 검증 없이 프로덕션 배포. 첫 이상 동작 시 원인 파악과 대응이 어렵습니다.

④ 트레이싱 미설정

운영 중 에이전트 동작 추적 불가. 문제 발생 시 재현도 원인 파악도 불가능합니다.

⑤ 무제한 루프

max_turns 미설정. 도구가 계속 실패하면 무한루프에 빠져 비용이 폭증합니다.

⑥ 컨텍스트 과프롬프트화

사용자 정보, 권한 등을 모두 instructions에 넣음. 보안 위험과 토큰 낭비.

⑦ 세션 없는 멀티턴

to_input_list 없이 단발 호출 반복. 에이전트가 매번 기억을 잃습니다.

⑧ 조기 멀티에이전트화

단일 에이전트로 해결 가능한 것을 복수 에이전트로 분리. 불필요한 복잡성 증가.

⑨ 오류를 RAISE로 처리

도구에서 예외를 raise하면 에이전트 루프가 중단됩니다. 오류 메시지를 반환하세요.

⑩ HITL 없는 액션 자동화

이메일, 결제, 파일 삭제 등 돌이킬 수 없는 행동에 사람 승인 없이 자동 실행.

L. 실전 설계 관점

기업 AX 도입 체크리스트

기업에서 Agents SDK 기반 에이전트 시스템을 도입하기 전에 확인해야 할 항목들입니다. 기술적 준비도와 운영 준비도 모두 포함됩니다.

기술 준비도

- ☐ Python 3.10+ 환경 확보
- ☐ OpenAI API 키 발급 및 예산 설정
- ☐ 에이전트가 접근할 내부 시스템 API 문서화
- ☐ 세션 저장소 선택 및 인프라 준비 (SQLite/Redis/RDB)
- ☐ Tracing 수집 대상 및 저장 정책 결정
- ☐ 가드레일 규칙 정의 (금지 주제, 권한 경계)
- ☐ HITL이 필요한 액션 목록 정의
- ☐ max_turns, 타임아웃 등 안전장치 설정

운영 준비도

- ☐ 에이전트 응답 품질 평가 기준 수립
- ☐ 이상 동작 알림 체계 구성
- ☐ 개인정보·민감정보 처리 방침 수립
- ☐ 에이전트 응답 로그 보존 기간 정책
- ☐ 모델 버전 업그레이드 테스트 절차
- ☐ 에이전트 실패 시 폴백 플로우 설계
- ☐ 사용자 피드백 수집 및 반영 프로세스
- ☐ 담당자 교육 및 운영 매뉴얼 작성

M. 사례형 슬라이드

코드 예시

실제 동작하는 PYTHON 예시 모음

Quickstart · Tool 추가 · Handoff · Session · Guardrail · Tracing

M. 사례형 슬라이드

예시 1: PYTHON QUICKSTART — 가장 기본적인 에이전트

설치부터 첫 실행까지. 가장 단순한 형태의 에이전트를 실행해봅니다. 이 예시는 도구 없이 텍스트만 처리하는 기본 에이전트입니다.

```python
# 1. 설치
# pip install openai-agents

import asyncio
from agents import Agent, Runner

# 2. 에이전트 정의
agent = Agent(
    name="assistant",
    instructions="""당신은 친절한 AI 어시스턴트입니다.
    한국어로 명확하고 간결하게 답변합니다.""",
    model="gpt-4o-mini",  # 비용 절감을 위해 mini 사용
)

# 3. 실행
async def main():
    result = await Runner.run(agent, "파이썬의 장점 3가지만 알려줘")
    print(result.final_output)

asyncio.run(main())
```

코드 해설

- `Agent()`: 설정 객체. 실행 시 상태 없음
- `Runner.run()`: 비동기 실행. 완료 후 결과 반환
- `result.final_output`: 최종 텍스트 응답
- 환경변수 `OPENAI_API_KEY` 필요

동기 환경에서 실행하려면

```python
# Jupyter, 스크립트 환경
result = Runner.run_sync(
    agent,
    "파이썬의 장점 3가지만 알려줘"
)
print(result.final_output)
```

M. 사례형 슬라이드

예시 2: TOOL 추가 – PYTHON 함수를 도구로 등록하기

에이전트에 외부 기능을 추가합니다. 날씨 조회 도구를 등록하고, 에이전트가 자동으로 도구를 호출하는 흐름을 확인합니다.

```python
from agents import Agent, Runner, function_tool
import asyncio

# 1. 도구 함수 정의
@function_tool
def get_weather(city: str) -> str:
    """지정된 도시의 현재 날씨를 조회합니다.

    Args:
        city: 날씨를 조회할 도시 이름 (한국어 또는 영어)
    """
    # 실제로는 기상청 API 호출
    weather_data = {
        "서울": "맑음, 22°C",
        "부산": "흐림, 19°C",
        "제주": "비, 18°C",
    }
    return weather_data.get(city, f"{city}의 날씨 정보를 찾을 수 없습니다.")

# 2. 도구를 가진 에이전트 정의
agent = Agent(
    name="weather-agent",
    instructions="날씨를 물어보면 반드시 get_weather 도구를 사용해 조회하고 답변합니다.",
    tools=[get_weather],
)

# 3. 실행 및 도구 호출 확인
async def main():
    result = await Runner.run(agent, "서울 날씨 어때?")
    print(result.final_output)
    # 도구 호출 이력 확인
    for item in result.new_items:
        print(f"[아이템] {item.type}: {item}")

asyncio.run(main())
```

핵심: 개발자는 루프를 구현하지 않았습니다. Runner가 자동으로 모델 호출 → 도구 파싱 → 함수 실행 → 결과 재입력 → 최종 응답 생성을 처리했습니다.

M. 사례형 슬라이드

예시 3: HANDOFF – 전문 에이전트로 대화 이전하기

사용자 문의 유형에 따라 적절한 전문 에이전트로 제어권을 넘기는 Triage 패턴 예시입니다. 핸드오프가 발생하면 last_agent가 변경된 것을 확인할 수 있습니다.

```
from agents import Agent, Runner, handoff
import asyncio

# 1. 전문 에이전트 정의
billing_agent = Agent(
    name="결제지원 에이전트",
    instructions="""결제, 환불, 청구서 관련 문의를 처리합니다.
    정확한 정보를 바탕으로 안내하고, 필요시 담당자 연결을 안내합니다.""",
)

tech_agent = Agent(
    name="기술지원 에이전트",
    instructions="""기술적 문제, 오류, 설치 관련 문의를 처리합니다.
    단계별로 문제를 진단하고 해결책을 제시합니다.""",
)

# 2. Triage 에이전트 정의
triage_agent = Agent(
    name="접수 에이전트",
    instructions="""고객 문의를 분류하여 적절한 전문 에이전트에게 연결합니다.
    - 결제/환불/청구 관련 → 결제지원 에이전트
    - 기술 문제/오류/설치 관련 → 기술지원 에이전트
    문의를 판단하기 어려우면 추가 질문을 합니다.""",
    handoffs=[billing_agent, tech_agent],
)

# 3. 실행 및 핸드오프 확인
async def main():
    result = await Runner.run(triage_agent, "결제가 두 번 됐는데 환불 받을 수 있나요?")
    print(f"최종 응답: {result.final_output}")
    print(f"마지막 에이전트: {result.last_agent.name}")
    # → 마지막 에이전트: 결제지원 에이전트

asyncio.run(main())
```

코드 해설: triage_agent가 "결제 환불"이라는 키워드를 감지하여 billing_agent로 핸드오프합니다. 이후 모든 처리는 billing_agent가 담당하며, result.last_agent.name이 "결제지원 에이전트"로 바뀐 것을 확인할 수 있습니다.

M. 사례형 슬라이드

예시 4: SESSION — 대화 이력을 유지하는 멀티턴 챗봇

세션을 활용하여 여러 턴에 걸친 대화 이력을 유지하는 예시입니다. SQLite 기반 세션 저장소를 사용하여 서버 재시작 후에도 대화가 이어집니다.

```python
from agents import Agent, Runner
from agents.sessions import SQLiteSession
import asyncio

agent = Agent(
    name="memory-agent",
    instructions="사용자와 자연스러운 대화를 이어갑니다. 이전 대화를 기억합니다.",
    model="gpt-4o-mini",
)

async def chat(session_id: str, user_message: str) -> str:
    """세션 ID로 대화를 이어갑니다."""
    session = SQLiteSession(
        session_id=session_id,
        db_path="./chat_history.db"
    )
    result = await Runner.run(
        agent,
        user_message,
        session=session,  # 세션 자동 로드 & 저장
    )
    return result.final_output

async def main():
    sid = "user_yms_001"

    # 첫 번째 턴
    r1 = await chat(sid, "안녕하세요, 저는 유민수입니다.")
    print(f"에이전트: {r1}")

    # 두 번째 턴 — 세션 덕분에 이름 기억
    r2 = await chat(sid, "제 이름 기억하시나요?")
    print(f"에이전트: {r2}")
    # → "네, 유민수 님이시죠!"

asyncio.run(main())
```

세션 동작 원리

- 첫 호출 시 세션 생성 및 DB에 저장
- 두 번째 호출 시 DB에서 이력 자동 로드
- 새 응답이 생성되면 자동으로 저장
- session_id가 같으면 항상 같은 이력

TO_INPUT_LIST 방식과 비교

- 세션 방식: 영속성 자동 처리, 코드 단순
- to_input_list: 메모리에만 유지, 직접 관리
- 웹 서비스라면 세션 방식이 훨씬 실용적

M. 사례형 슬라이드

예시 5: GUARDRAIL & HITL — 안전 장치 구현하기

금지 주제를 차단하는 Input Guardrail과, 민감한 도구 실행 전 사람 승인을 요청하는 흐름을 구현합니다.

```python
from agents import (Agent, Runner, input_guardrail,
                    GuardrailFunctionOutput, function_tool)
from agents import RunContextWrapper
import asyncio

# 1. Input Guardrail: 금지 주제 차단
@input_guardrail
async def block_off_topic(ctx, agent, input_data) -> GuardrailFunctionOutput:
    """업무 외 주제를 차단합니다."""
    forbidden = ["주식", "도박", "정치"]
    text = str(input_data).lower()
    triggered = any(word in text for word in forbidden)
    return GuardrailFunctionOutput(
        output_info={"reason": "off-topic"} if triggered else {},
        tripwire_triggered=triggered,
    )

# 2. 민감 도구 — 실행 전 로그 기록
@function_tool
async def send_notification(ctx: RunContextWrapper, message: str, channel: str) -> str:
    """알림을 발송합니다. channel: 'slack' 또는 'email'

    Args:
        message: 발송할 메시지 내용
        channel: 발송 채널 (slack 또는 email)
    """
    # 실제로는 Slack/Email API 호출
    print(f"[알림 발송] {channel}: {message}")
    return f"{channel}으로 알림이 발송되었습니다."

# 3. 가드레일이 적용된 에이전트
agent = Agent(
    name="safe-notifier",
    instructions="업무 관련 알림 발송을 돕는 에이전트입니다.",
    tools=[send_notification],
    input_guardrails=[block_off_topic],
)

async def main():
    # 정상 요청
    result = await Runner.run(agent, "팀 채널에 배포 완료 알림 보내줘")
    print(result.final_output)

    # 차단되는 요청
    try:
        result = await Runner.run(agent, "주식 투자 어떻게 해?")
    except Exception as e:
        print(f"차단됨: {e}")

asyncio.run(main())
```

코드 해설: block_off_topic 가드레일이 "주식" 키워드를 감지하면 tripwire_triggered=True를 반환하고, Runner는 즉시 실행을 중단하여 예외를 발생시킵니다. 메인 에이전트 실행이 시작되지도 않습니다.

M. 사례형 슬라이드

예시 6: TRACING – 실행 흐름 추적하고 커스텀 처리하기

Tracing을 활용하여 에이전트 실행을 추적하고, 커스텀 프로세서로 로컬 파일에 저장하는 예시입니다.

```python
from agents import Agent, Runner
from agents.tracing import add_trace_processor, TracingProcessor
from agents.tracing.spans import Span
import asyncio, json, datetime

# 1. 커스텀 트레이싱 프로세서
class FileTraceProcessor(TracingProcessor):
    def __init__(self, filepath: str):
        self.filepath = filepath

    def on_span_end(self, span: Span) -> None:
        """스팬이 종료될 때마다 파일에 기록"""
        record = {
            "timestamp": datetime.datetime.now().isoformat(),
            "trace_id": span.trace_id,
            "span_type": span.span_data.type,
            "duration_ms": span.duration_ms,
        }
        with open(self.filepath, "a") as f:
            f.write(json.dumps(record, ensure_ascii=False) + "\n")

# 2. 프로세서 등록
add_trace_processor(FileTraceProcessor("agent_traces.jsonl"))

# 3. 에이전트 실행 — 자동으로 trace 수집됨
agent = Agent(
    name="traced-agent",
    instructions="질문에 답변합니다.",
    model="gpt-4o-mini",
)

async def main():
    result = await Runner.run(agent, "Agents SDK의 핵심 장점은?")
    print(result.final_output)
    print("트레이스가 agent_traces.jsonl에 저장되었습니다.")

asyncio.run(main())
```

프로덕션에서 유용한 패턴

- Langfuse, Datadog으로 스팬 전송
- 실패한 스팬만 Slack 알림
- 토큰 사용량 집계 및 비용 모니터링
- 특정 도구 호출 빈도 통계 수집

민감정보 마스킹 예시

```python
# 민감 데이터 캡처 비활성화
from agents import set_tracing_export_api_key
from agents.tracing import TRACE_INCLUDE_SENSITIVE

# 환경변수로 제어
# OPENAI_AGENTS_DONT_LOG_TOOL_DATA=1
```

M. 사례형 슬라이드

예시 7: 구조화된 출력 – OUTPUT_TYPE으로 JSON 응답 강제하기

에이전트가 자유 텍스트 대신 정해진 구조의 객체를 반환하도록 강제합니다. 다운스트림 시스템과 연동할 때 필수적인 패턴입니다.

```python
from agents import Agent, Runner
from pydantic import BaseModel, Field
import asyncio

# 1. 출력 스키마 정의 (Pydantic)
class TicketClassification(BaseModel):
    category: str = Field(description="문의 카테고리: billing/technical/general")
    priority: str = Field(description="우선순위: high/medium/low")
    summary: str = Field(description="문의 내용 한 줄 요약 (50자 이내)")
    requires_human: bool = Field(description="사람 처리가 필요한지 여부")

# 2. output_type으로 구조화 출력 강제
classifier = Agent(
    name="ticket-classifier",
    instructions="""고객 문의를 분석하여 분류합니다.
    - category: billing(결제), technical(기술), general(일반)
    - priority: 긴급도에 따라 high/medium/low 판단
    - requires_human: 복잡하거나 민감한 경우 True""",
    output_type=TicketClassification,  # Pydantic 모델 지정
    model="gpt-4o-mini",
)

async def main():
    result = await Runner.run(
        classifier,
        "어제부터 로그인이 안 됩니다. 중요한 발표가 오늘인데 빨리 해결해 주세요."
    )
    ticket: TicketClassification = result.final_output
    print(f"카테고리: {ticket.category}")      # technical
    print(f"우선순위: {ticket.priority}")      # high
    print(f"요약: {ticket.summary}")
    print(f"사람 처리 필요: {ticket.requires_human}")

asyncio.run(main())
```

실무 포인트: `output_type`을 사용하면 SDK가 내부적으로 structured output 모드를 활성화합니다. 에이전트 응답을 dict로 파싱하는 코드를 직접 작성할 필요가 없어지고, 타입 안전성도 확보됩니다.

N. 마무리

마무리

핵심 요약과 학습 로드맵

전체 요약 · 학습 순서 · 앞으로의 Agent Engineering · 기업 도입 질문

N. 마무리

전체 핵심 요약: 이것만 기억하면 된다

70장 분량의 내용을 7가지 핵심 메시지로 압축했습니다. 이 7가지를 이해했다면 Agents SDK의 본질을 파악한 것입니다.

1 AGENTS SDK는 에이전트 런타임이다

단순 챗봇 라이브러리가 아닙니다. 루프, 도구 실행, 상태 관리, 핸드오프, 가드레일을 오케스트레이션하는 **실행 엔진**입니다. Responses API 위의 계층입니다.

2 AGENT는 설정, RUNNER는 실행이다

Agent 객체는 "무엇을 할 수 있는가"를 선언합니다. Runner가 "어떻게 실행할 것인가"를 담당합니다. 이 역할 분리가 SDK 설계의 핵심입니다.

3 도구의 품질이 에이전트의 품질이다

명확한 이름, 충분한 docstring, 예측 가능한 반환값. 도구 설계가 나쁘면 모델이 아무리 좋아도 에이전트는 실패합니다.

4 LLM 컨텍스트와 코드 컨텍스트를 분리하라

모델이 읽어야 할 것은 instructions와 세션에, 코드가 읽어야 할 것은 RunContextWrapper에 넣으세요. 섞으면 보안과 비용 모두 문제가 생깁니다.

5 단일 에이전트를 먼저 완성하라

멀티에이전트는 필요에 의해 분리하는 것이지, 처음부터 설계하는 것이 아닙니다. 단일 에이전트 + 도구 + 세션 + 가드레일이 먼저입니다.

6 TRACING 없는 프로덕션은 없다

에이전트가 왜 그런 결과를 냈는지 설명할 수 없다면 운영이 아닙니다. Tracing은 선택이 아닌 필수 인프라입니다.

7 돌이킬 수 없는 행동엔 반드시 HITL을 붙여라

자동화의 목적은 사람을 제거하는 것이 아니라 사람이 더 중요한 일에 집중하게 하는 것입니다. 위험한 액션에는 항상 사람 검토를 넣으세요.

N. 마무리

AGENTS SDK 학습 로드맵: 어떤 순서로 배우면 되는가

처음 접하는 학습자가 단계적으로 깊이를 쌓아갈 수 있는 권장 학습 경로입니다. 각 단계를 건너뛰지 말고 순서대로 익히는 것이 가장 효율적입니다.

1단계: 기초 실습 (1~2일)

환경 설치 → Agent + Runner.run_sync() → function_tool 하나 등록 → result.final_output 확인. 목표: "도구 하나 달린 에이전트를 직접 실행해본다"

2단계: 상태와 구조화 (3~5일)

Session + to_input_list → output_type(Pydantic) → RunContextWrapper 활용. 목표: "실제 서비스에 붙일 수 있는 수준의 에이전트를 만든다"

3단계: 제어와 안전 (1주)

Input/Output Guardrail → Tracing 설정 → max_turns, 오류 처리. 목표: "안전하게 운영할 수 있는 에이전트를 만든다"

4단계: 멀티에이전트 (2주 이상)

Handoff → Triage 패턴 → Manager + agents-as-tools → MCP 연결. 목표: "복잡한 업무 자동화 시스템을 설계할 수 있다"

N. 마무리

AGENT ENGINEERING의 다음 학습 포인트

Agents SDK를 익힌 후 에이전트 엔지니어링을 더 깊이 파고들고 싶다면 다음 주제들이 중요합니다. 이 분야는 빠르게 발전하고 있으며, 기반 개념을 탄탄히 해 두는 것이 중요합니다.

기술 심화 학습 주제

- **평가(Evals):** 에이전트 응답 품질을 측정하는 방법론. LLM-as-judge, 골든 데이터셋 구성
- **프롬프트 엔지니어링 심화:** 지시사항 작성 원칙, few-shot 예시 활용, 역할 정의
- **RAG 심화:** 청킹 전략, 임베딩 모델 선택, 하이브리드 검색
- **에이전트 보안:** 프롬프트 인젝션 방어, tool call 검증, 권한 최소화
- **비용 최적화:** 모델 선택, 캐싱 전략, 토큰 압축

생태계 연계 학습 주제

- **MCP 서버 제작:** 사내 시스템을 MCP 서버로 노출하는 방법
- **LangSmith / Langfuse:** 고급 트레이싱 및 평가 플랫폼 활용
- **FastAPI 통합:** 에이전트를 REST API 서비스로 배포하기
- **Vector DB:** Pinecone, Qdrant, pgvector 활용 RAG 구축
- **에이전트 패턴:** ReAct, Plan-and-Execute, Reflection 패턴

권장 참고 자료: OpenAI Agents SDK 공식 문서(https://openai.github.io/openai-agents-python/ko/), OpenAI Cookbook의 에이전트 예시, Anthropic의 에이전트 설계 가이드

N. 마무리

기업 AX 도입 시 반드시 질문해야 할 항목

기업에서 Agents SDK 기반 시스템 도입을 검토할 때, 기술팀과 의사결정자가 함께 논의해야 할 핵심 질문들입니다. 이 질문에 명확한 답이 없다면 도입을 서두르지 않는 것이 좋습니다.

📋 범위와 목적

- 이 에이전트가 자동화할 업무는 명확히 정의되어 있는가?
- 자동화 전과 후의 성공 기준(KPI)은 무엇인가?
- 사람이 반드시 검토해야 하는 단계는 어디인가?

🔒 보안과 데이터

- 에이전트가 접근할 데이터의 민감도는 어느 수준인가?
- 개인정보가 모델에 전달되는 경우 법적 검토가 완료되었는가?
- OpenAI API로 전송되는 데이터의 보안 정책을 확인했는가?

💰 비용과 운영

- 월간 예상 API 호출 수와 토큰 비용을 계산했는가?
- 에이전트 실패 시 폴백 프로세스가 있는가?
- 담당 운영자의 기술 역량과 교육 계획은?

🧪 검증과 품질

- 에이전트 응답 품질을 주기적으로 평가하는 방법이 있는가?
- 모델 업그레이드 시 회귀 테스트 절차가 있는가?
- 사용자 피드백을 수집하고 개선에 반영하는 루프가 있는가?

N. 마무리

AGENT SDK가 바꾸는 개발자의 역할

Agents SDK의 등장은 단순히 새로운 라이브러리가 추가된 것이 아닙니다. AI 시스템을 다루는 개발자와 기획자의 역할 자체가 변화하고 있습니다.

과거 (프롬프트 앱 시대)	현재 (에이전트 SDK 시대)	미래 (에이전트 플랫폼 시대)
프롬프트 작성이 핵심 역량	에이전트 아키텍처 설계가 핵심	에이전트 네트워크 오케스트레이션
모델 호출 1회로 결과 완성	루프, 도구, 상태를 설계	자율 에이전트 시스템 운영
응답 품질 = 프롬프트 품질	응답 품질 = 시스템 설계 품질	응답 품질 = 에이전트 생태계 품질
운영은 별도 팀 담당	Tracing으로 직접 운영 가시성 확보	자가 개선하는 에이전트 시스템

> 앞으로 AI 시스템을 잘 만드는 것은 "좋은 프롬프트를 쓰는 능력"보다 "에이전트 런타임을 설계하고 운영하는 능력"에 의해 결정될 것입니다. Agents SDK는 그 역량을 쌓는 첫 번째 언어입니다.

부록

자주 묻는 질문(FAQ) 10가지

Agents SDK를 처음 배우는 사람들이 가장 자주 묻는 질문과 답변을 정리했습니다.

Q1. AGENTS SDK는 유료인가요?

SDK 자체는 오픈소스(MIT)이며 무료입니다. 단, 내부적으로 OpenAI API를 호출하므로 API 사용 요금이 발생합니다.

Q2. GPT-4O 외 다른 모델도 쓸 수 있나요?

네. Chat Completions 호환 API라면 LiteLLM을 통해 Claude, Gemini 등도 사용 가능합니다. 단, Hosted Tools는 OpenAI 전용입니다.

Q3. 에이전트 루프가 무한히 돌 수 있나요?

기본적으로 max_turns 제한이 있으며, 초과 시 MaxTurnsExceeded 예외가 발생합니다. 반드시 적절한 값으로 설정하세요.

Q4. LANGCHAIN과 함께 쓸 수 있나요?

기술적으로 불가능하지는 않지만, 두 프레임워크의 추상화가 겹쳐 복잡성이 크게 증가합니다. 하나를 선택하여 일관되게 사용하는 것을 권장합니다.

Q5. 에이전트가 잘못된 도구를 선택하면 어떻게 하나요?

도구의 이름과 docstring을 더 명확하게 수정하거나, instructions에서 도구 선택 기준을 명시하세요. 도구 수를 줄이는 것도 효과적입니다.

Q6. 스트리밍과 세션을 함께 쓸 수 있나요?

네. `Runner.run_streamed()`에 `session=` 인자를 전달하면 됩니다. 스트리밍 중에도 세션이 자동으로 업데이트됩니다.

Q7. 도구 실행 결과가 너무 길면 어떻게 되나요?

토큰 한도를 초과할 수 있습니다. 도구 함수 내에서 결과를 요약하거나 청크로 나누어 반환하는 것을 권장합니다.

Q8. 비동기(ASYNC)를 꼭 써야 하나요?

아닙니다. `Runner.run_sync()`를 사용하면 동기 환경에서도 실행 가능합니다. 프로덕션 서비스라면 async가 성능상 유리합니다.

Q9. 에이전트를 DOCKER 컨테이너에서 실행할 수 있나요?

네. 일반 Python 애플리케이션과 동일하게 컨테이너화 가능합니다. 환경변수로 API 키와 세션 저장소 연결 정보를 주입하면 됩니다.

Q10. SDK 버전 업그레이드 시 주의할 점은?

Agent, Runner, Tool의 인터페이스는 비교적 안정적이지만, 내부 구현이 바뀔 수 있습니다. 버전을 고정(`openai-agents==x.y.z`)하고 업그레이드 전 통합 테스트를 수행하세요.

부록

용어 정리: AGENTS SDK 핵심 용어 사전

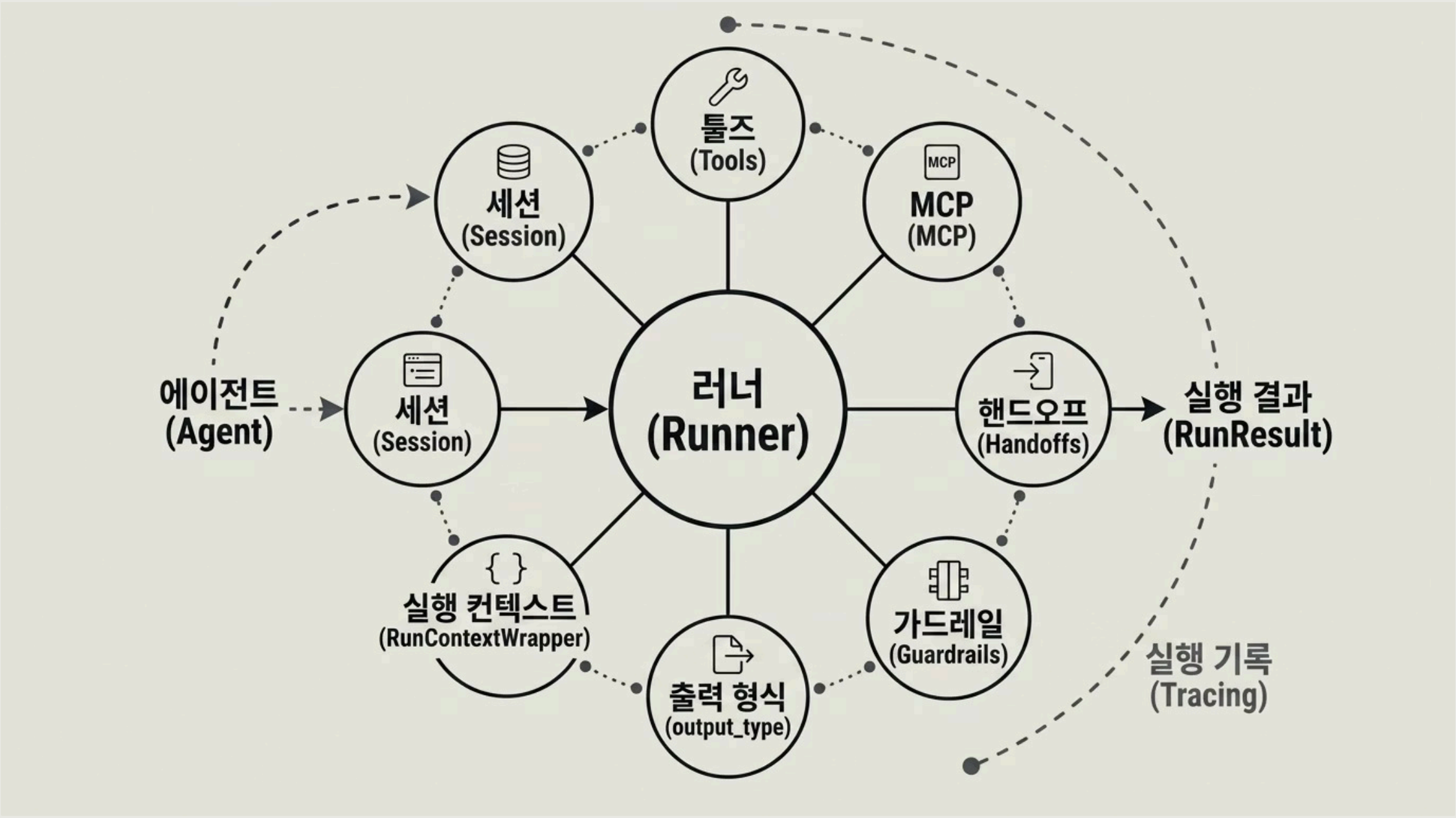
이 덱에서 자주 등장한 용어들을 한곳에 정리했습니다. 처음 접한 용어가 있다면 이 페이지를 참고하세요.

용어	정의
Agent	이름, 지시사항, 모델, 도구, 핸드오프를 가진 실행 단위. 설정 객체이며 Runner가 실행.
Runner	Agent를 실행하는 루프 엔진. run(), run_sync(), run_streamed() 제공.
RunResult	Runner.run() 실행 완료 후 반환되는 결과 객체. final_output, new_items, last_agent 포함.
function_tool	Python 함수를 에이전트 도구로 변환하는 데코레이터. 함수 시그니처에서 JSON 스키마 자동 생성.
Handoff	한 에이전트에서 다른 에이전트로 대화 제어권을 완전히 이전하는 메커니즘.
Guardrail	입력/출력/도구 호출 단계에서 검증을 수행하는 안전 계층.
Tripwire	가드레일이 위반을 감지했을 때 발동하여 에이전트 실행을 즉시 중단하는 메커니즘.
Session	대화 이력을 구조화된 저장소에 유지하는 메커니즘. SQLite, Redis, 메모리 등 지원.
RunContextWrapper	LLM에 노출되지 않는 코드 런타임 컨텍스트. 사용자 정보, DB 클라이언트 등을 도구에 주입.
Trace	하나의 Runner.run() 호출 전체를 나타내는 실행 기록의 루트 노드.
Span	Trace 내의 개별 작업 단위. 모델 호출, 도구 실행, 핸드오프 등이 각각 Span.
MCP	Model Context Protocol. 외부 시스템을 표준화된 인터페이스로 에이전트에 연결하는 프로토콜.
HITL	Human-in-the-Loop. 에이전트가 민감한 액션을 취하기 전 사람의 승인을 요청하는 패턴.
output_type	에이전트의 최종 출력을 Pydantic 모델로 강제하는 속성. 구조화된 JSON 출력을 보장.
tool_use_behavior	도구 실행 후 루프 동작 제어. run_llm_again(기본) 또는 stop_on_first_tool 등 선택.

부록

AGENTS SDK 구성요소 관계도: 전체 한눈에 보기

이 덱에서 다룬 모든 구성요소가 어떻게 연결되는지 마지막으로 정리합니다.



기억할 것: Runner가 중심입니다. Agent는 Runner에게 설정을 제공하고, Context와 Session은 실행 환경을 주입합니다. Tools와 Handoffs는 루프 안에서 동작하며, Guardrails는 각 단계를 감시합니다. Tracing은 이 모든 것을 기록합니다.

부록

빠른 참조: 실행 메서드와 주요 API 치트시트

개발 중 자주 참조하게 되는 핵심 API를 한 페이지에 정리했습니다. 북마크해두고 활용하세요.

AGENT 생성

```
Agent(
  name="이름",
  instructions="지시사항 또는 함수",
  model="gpt-4o",
  model_settings=ModelSettings(
    temperature=0.3,
    max_tokens=1000,
  ),
  tools=[my_tool],
  handoffs=[other_agent],
  output_type=MyPydanticModel,
  input_guardrails=[my_guardrail],
  output_guardrails=[my_output_guard],
)
```

RUNNER 실행

```
# 비동기
result = await Runner.run(agent, input)

# 동기
result = Runner.run_sync(agent, input)

# 스트리밍
stream = Runner.run_streamed(agent, input)
async for event in stream.stream_events():
    ...
```

결과 활용

```
result.final_output      # 최종 응답
result.last_agent        # 마지막 에이전트
result.new_items         # 실행 중 생성된 아이템
result.to_input_list()   # 다음 턴 입력으로 변환
```

세션 & 컨텍스트

```
# 세션
session = SQLiteSession(
    session_id="user_123",
    db_path="chat.db"
)
await Runner.run(agent, msg, session=session)

# 컨텍스트
await Runner.run(agent, msg, context=MyContext(...))
```

도구 등록

```
@function_tool
def my_tool(param: str) -> str:
    """도구 설명 (모델이 읽음)"""
    return "결과"
```

OPENAI AGENTS SDK: 학습을 마치며

이 덱을 끝까지 읽으셨다면 Agents SDK의 전체 구조를 이해하고,
실제 시스템 설계에 적용할 수 있는 기반을 갖추셨습니다.

🎯 다음 액션 1

오늘 배운 내용으로 간단한 에이전트를 직접 만들어보세요. 도구 하나, 세션 하나부터 시작합니다.

📖 다음 액션 2

공식 문서(openai.github.io/openai-agents-python/ko/)의 예시 코드를 실행해보고 이 덱의 내용과 연결해보세요.

🏢 다음 액션 3

기업 도입을 고려한다면 도입 체크리스트(L 섹션)를 팀과 함께 점검하고, 파일럿 프로젝트 범위를 정의하세요.

> "에이전트 시스템의 복잡성을 두려워하지 마세요. 단일 에이전트 + 도구 하나의 조합만으로도 현업의 많은 문제를 해결할 수 있습니다. 완벽한 설계보다 작동하는 시스템이 먼저입니다."

감사합니다

질문이 있으시면 언제든지 연락주세요

기업 AX 도입 · AI 개발 솔루션 · AI 교육 문의

귀사의 업무에 맞는 에이전트 시스템 설계부터 구현, 교육까지 함께합니다.

- 기업 AI 전환(AX) 전략 수립 및 파일럿 구축
- OpenAI Agents SDK 기반 솔루션 개발
- 개발자 · 기획자 대상 AI 기술 교육
- 내부 업무 자동화 에이전트 제작

연락처

개발자

유민수

📞 전화

010-2773-2165

🌐 웹사이트

WWW.DEFORMATIC.AI.KR

참고

참고 자료 및 출처

이 덱을 작성하는 데 참고한 공식 자료와 추천 학습 자료입니다.

공식 문서

- OpenAI Agents SDK 공식 한국어 문서
  openai.github.io/openai-agents-python/ko/
- OpenAI Responses API 문서
  platform.openai.com/docs/api-reference/responses
- OpenAI Cookbook – 에이전트 예시
  cookbook.openai.com
- Model Context Protocol 공식 사이트
  modelcontextprotocol.io

추천 학습 자료

- Anthropic – Building Effective Agents
  에이전트 설계 원칙과 패턴 가이드
- OpenAI – Agents SDK GitHub Repository
  github.com/openai/openai-agents-python
- Langfuse 공식 문서 – Tracing 연동
  langfuse.com/docs
- Pydantic 공식 문서 – output_type 활용
  docs.pydantic.dev

이 덱의 내용은 OpenAI Agents SDK 공식 문서를 바탕으로 교육 목적에 맞게 재구성되었습니다. SDK는 활발히 업데이트되고 있으므로 최신 변경사항은 공식 문서에서 확인하세요. 작성 기준: 2025년 상반기 기준 공식 문서.