

WEEK 01

함께하는 AI 에이전트 개발

1주차: 대화형 AI 코어 구축



PROJECT GOAL

리서치 어시스턴트



DURATION

180 Mins (3h)



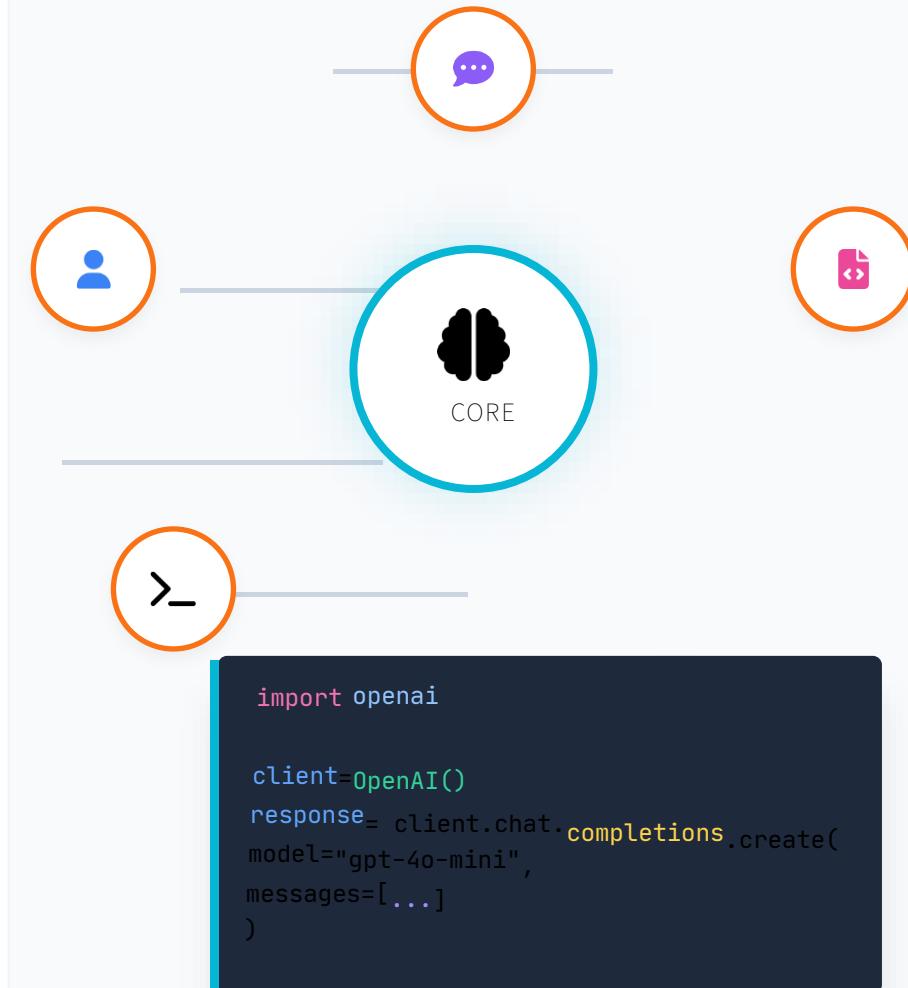
CORE STACK

Python, OpenAI API



INSTRUCTOR

Thomas Moon



A 학습 로드맵



TOTAL: 15.5 HOURS



학습 목표

이번 과정을 통해 달성할 핵심 가치입니다.

-  핵심 개념 이해. AI 에이전트의 작동 원리와 아키텍처를 깊이 있게 이해합니다.
-  실전 프로젝트. 실무에 즉시 적용 가능한 '리서치 어시스턴트'를 개발합니다.
-  체계적 학습. 기초부터 심화 기능까지 점진적으로 기능을 확장합니다.

01

SECTION

PREVIEW FINAL PROJECT

최종 결과물 미리보기

이번 과정을 통해 완성하게 될 'AI 리서치 어시스턴트'의 주요 기능과 실제 활용 모습을 미리 확인해봅니다.

★ WHAT TO EXPECT

🔍 자동 웹 정보 수집 및 분석

📋 구조화된 리포트 생성

🗣 자연스러운 대화 인터페이스

📊 자율적 판단과 도구 사용

AI Research Assistant란?

정보 수집부터 리포트 작성까지 자동화

프로젝트 핵심 개념

단순한 검색 도구가 아닌, 스스로 판단하고 실행하는 지능형
에이전트 시스템입니다.

1 Input Processing

사용자의 질문이나 모호한 주제를 이해하고 구체적인 검색 목표로
변환합니다.

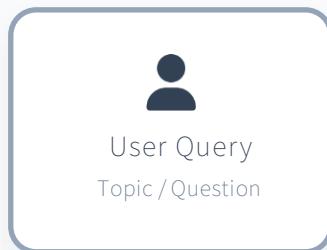
2 Autonomous Research

웹에서 정보를 자동 수집하고 여러 소스를 교차 검증하여 신뢰도
를 높입니다.

3 Structured Output

수집된 정보를 종합하여 사용자가 원하는 형식의 전문적인 리포트
를 자동 생성합니다.

WORKFLOW DIAGRAM



완성 시스템의 핵심 기능

8가지 핵심 기능 명세

01



자연스러운 대화 인터페이스

사용자와 자연어로 소통하며 의도를 파악하고 맥락을 유지

02



자동 웹 검색 및 정보 수집

실시간 웹 검색을 통해 최신 정보를 자동으로 수집하고 필터링

03



벡터 DB 기반 지식 관리

수집된 정보를 임베딩하여 영구 저장하고 의미 기반 검색 지원

04



ReAct 패턴 자율 실행

Reasoning(추론)과 Acting(행동) 루프를 통해 스스로 판단하고 실행

05



작업 분해 및 계획

복잡한 작업을 하위 단계로 분해하고 순차적으로 수행 계획 수립

06



멀티 에이전트 협업

검색, 분석, 작문 등 각 분야의 전문 에이전트가 협업하여 작업 수행

07



자동 품질 검증

생성된 결과물의 정확성과 완성도를 스스로 평가하고 수정

08



전문적인 리포트 생성

수집된 정보를 바탕으로 구조화된 마크다운 형식의 리포트 자동 작성

실무 활용 사례

다양한 도메인에서의 확장성

도메인별 특화 기능

AI 에이전트는 검색 목적과 출력 형식을 커스터마이징하여 다양한 산업 분야에 즉시 적용 가능합니다.

마케팅 (Marketing)

시장 트렌드와 소비자 반응을 실시간으로 수집하고 분석하여 인사이트를 도출합니다.

투자 분석 (Investment)

기업 실적 발표와 뉴스, 산업 리포트를 종합하여 투자 의사결정을 지원합니다.

기술 리서치 (Tech)

최신 기술 스택 비교, 오픈소스 동향, 특히 분석 등 기술적 심층 조사를 수행합니다.

컨설팅 (Consulting)

클라이언트 산업군에 대한 방대한 자료 조사와 베스트 프랙티스 수집을 자동화합니다.

MULTI-DOMAIN APPLICATION

Marketing

시장 트렌드 분석
경쟁사 조사 자동화
소비자 인사이트

Investment

기업 실적 분석
산업 동향 리서치
의사결정 지원



AI CORE

Tech R&D

최신 기술 동향
특히 분석
기술 스택 비교

Consulting

클라이언트 산업 조사
베스트 프랙티스
제안서 작성 지원

5주 개발 로드맵

총 소요시간: **15.5 Hours**

W1



대화형 AI 코어

기본 대화 시스템 구축 및 프롬프트 엔지니어링

⌚ 3.0h

W2



정보 수집 에이전트

웹 검색 API 연동 및 외부 도구(Tools) 사용

⌚ 3.0h

W3



메모리 시스템

RAG 아키텍처 및 벡터 DB를 활용한 장기 기억

⌚ 3.0h

W4



자율 실행 (ReAct)

스스로 계획을 수립하고 추론하는 ReAct 패턴 구현

⌚ 3.0h

W5



멀티 에이전트

여러 전문 에이전트 간의 협업 시스템 및 리포트 생성

⌚ 3.5h

ALL



과정 수료 및 완성

5주간의 실습을 통해 나만의 AI 비서 완성

☒ Total 15.5h

오늘의 학습 목표

MAIN OBJECTIVE

자연스러운 대화가 가능한 리서치 어시스턴트 기본 틀 완성



AI 에이전트의 개념과 구조 이해

THEORY



OpenAI API 연동 및 활용

PRACTICE



대화 히스토리 관리 시스템 구현

LOGIC



전문적인 페르소나를 가진 어시스턴트 구축

DESIGN

이번 시간의 핵심 달성 과제입니다.



Research Assistant

오늘 우리는 스스로 생각하고 대화하는 나만의 AI 비서 MVP를 만듭니다.

3

HOURS

4

MISSIONS

1

PROJECT

02

SECTION

THEORY

AI FUNDAMENTALS

AI 에이전트의 이해

AI 에이전트의 개념부터 아키텍처, 핵심 기술인 LLM과 프롬프트 엔지니어링까지
심층적으로 탐구합니다.

KEY TOPICS

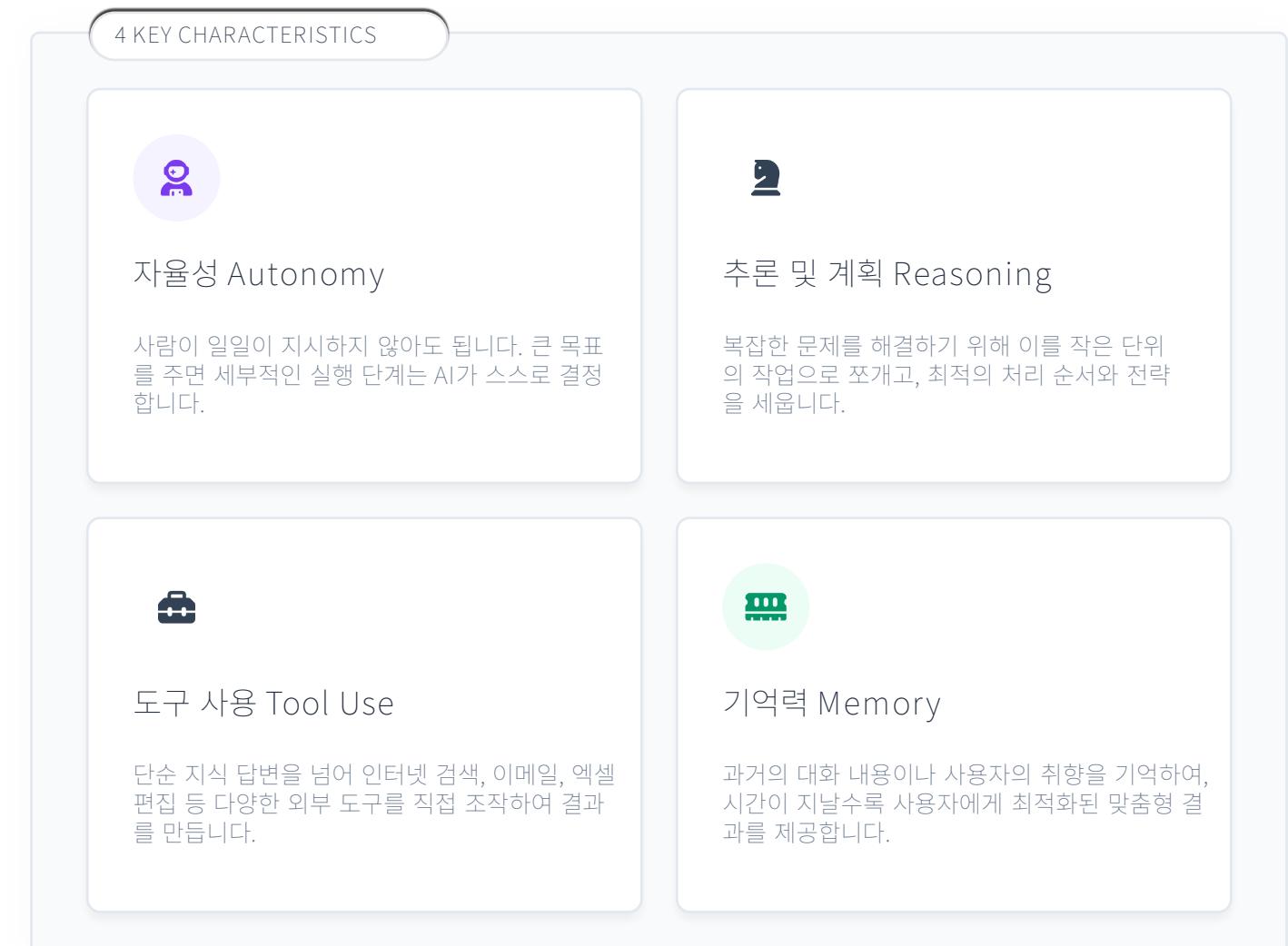
- 🤖 AI 에이전트란 무엇인가?
- 📦 프로젝트 아키텍처
- ✍️ 프롬프트 엔지니어링
- 🕸️ 핵심 구성요소
- ☁️ LLM API 생태계

AI 에이전트란?

단순 응답을 넘어 목표를 완수하는 시스템

행동하는 AI

일반적인 AI는 정보 제공에 그치지만, AI 에이전트는 복잡한 명령을 수행하기 위해 도구를 사용하고 스스로 행동 과정을 결정합니다.



COMPARISON

전통적 챗봇 vs AI 에이전트

규칙 기반에서 자율 실행으로의 진화

 동작 방식 BOT 규칙 기반, 사전 정의된 스크립트 AGENT 상황에 따른 자율적 판단 및 실행	 응답 범위 BOT 준비된 답변만 출력 가능 AGENT 정보를 종합하여 실시간 동적 생성	 도구 사용 BOT 매우 제한적 기능 연동 AGENT 검색, API, 코드 등 자유로운 활용
 학습 능력 BOT 없음 (정적 데이터베이스) AGENT 컨텍스트 학습 및 지속적 개선	 복잡한 작업 BOT 불가능 (단답형 처리) AGENT 작업 분해(Decomposition) 후 수행	 실제 예시 BOT FAQ 봇, ARS, 주문 키오스크 AGENT 리서치 어시스턴트, 코딩 파트너

AI Agent 아키텍처와 요소

에이전트 시스템의 작동 원리

Cognitive Architectures

에이전트가 정보를 인식하고, 추론하며, 행동을 결정하는 전체적인 인지 구조를 의미합니다.

Orchestration (제어 센터)

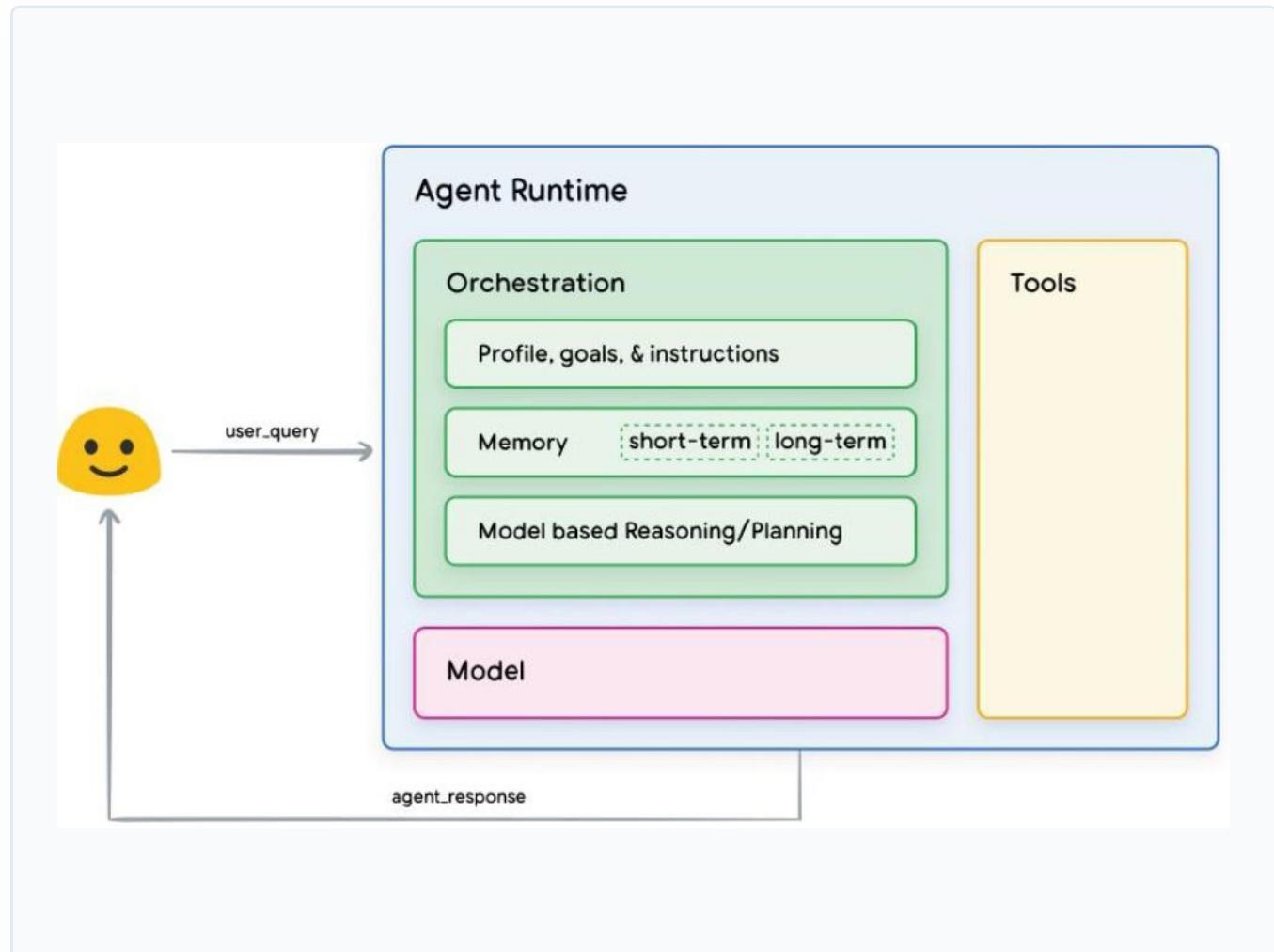
프로필, 목표, 지침을 관리하고 Memory(단기/장기 기억)를 활용하여 추론(Reasoning) 및 계획(Planning)을 수립합니다.

Model (두뇌)

LLM을 기반으로 언어를 이해하고 의사결정을 내리는 핵심 엔진입니다. Orchestration 레이어의 지시를 따릅니다.

Tools (외부 인터페이스)

API, 검색 엔진, 데이터베이스 등 에이전트가 외부 세계와 상호작용하기 위해 사용하는 도구 집합입니다.



Agent 구성 요소 - 모델(Model)

LLM 기반 의사결정 엔진

에이전트의 두뇌 역할

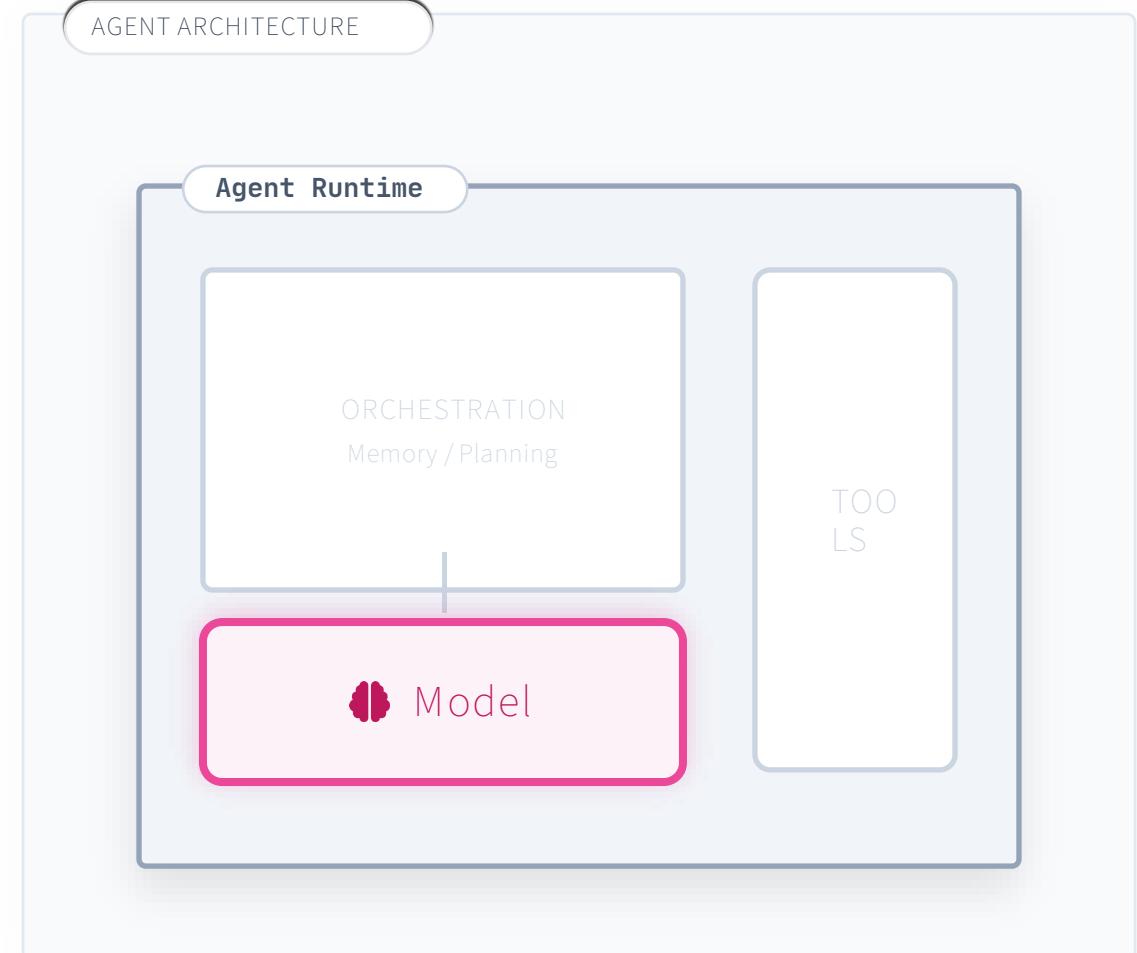
모델은 LLM(Large Language Model)으로 구성되며, 프롬프트 기반 추론과 논리를 바탕으로 의사결정을 수행합니다. 사용자의 질문이나 목표, 기억된 정보를 종합해 다음 행동을 결정합니다.

❖ 다양한 모델 형태

- 범용 LLM: GPT-4, Gemini, Claude 3 등 일반적인 추론 능력 보유
- 멀티모달 모델: 텍스트 외에도 이미지, 음성 등 다양한 데이터 처리
- 파인튜닝 모델: 특정 도메인이나 목적에 맞춰 미세 조정된 모델

❗ 중요한 점

모델은 일반적으로 에이전트의 도구 설정을 직접 학습하지 않습니다. 문맥에 맞춰 어떤 도구를 사용할지 결정하는 '판단력'을 제공합니다. 에이전트의 성능은 모델의 추론 능력과 올바른 도구를 선택하는 능력에 가장 크게 좌우됩니다.



Orchestration Layer

Agent의 제어 센터 (Control Center)

• 에이전트의 지휘자

사용자의 입력을 해석하고, 목표를 달성하기 위해 어떻게 생각(추론)하고 행동할지 결정하는 두뇌 역할을 수행합니다.

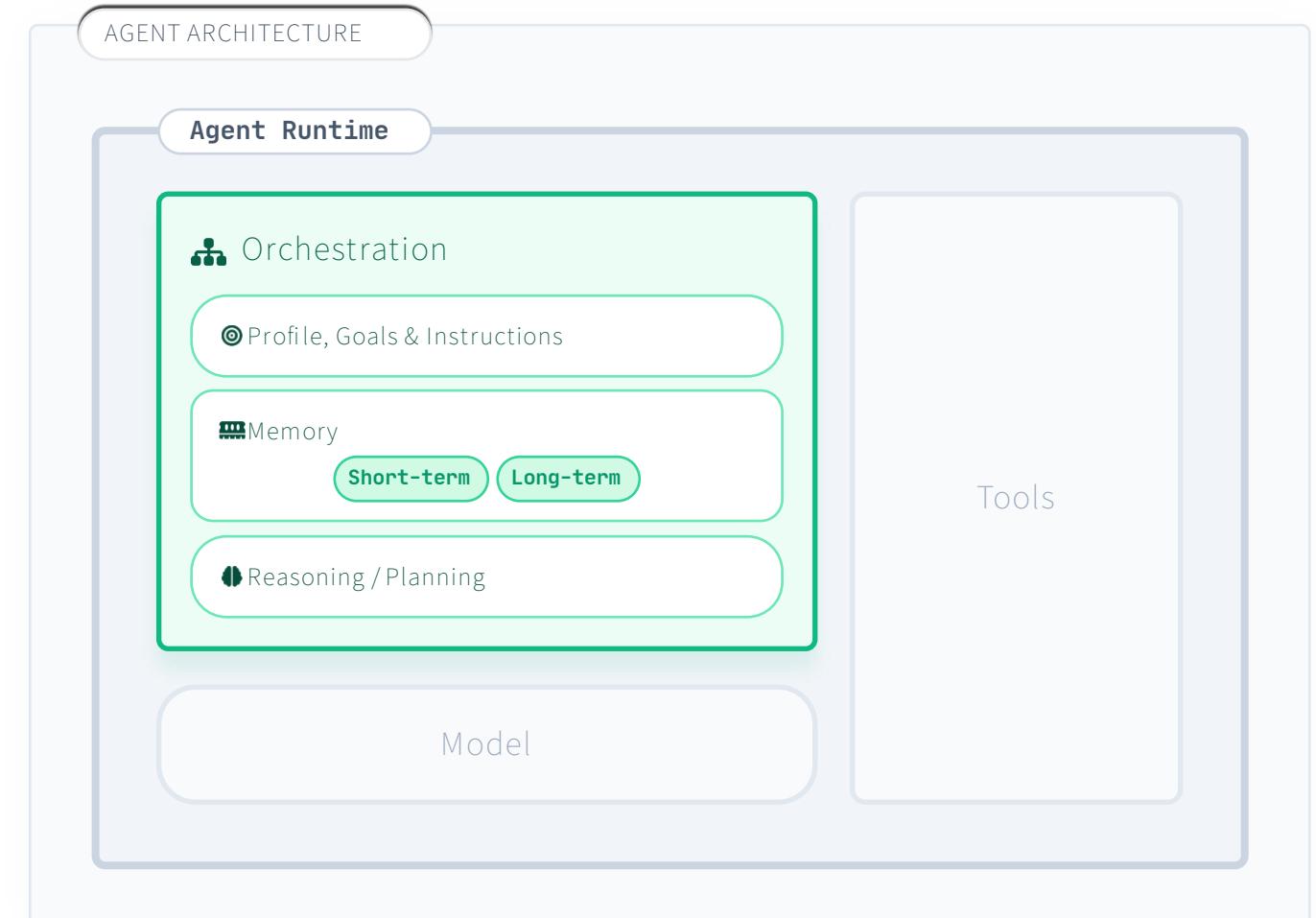
• 주요 기능 & 루프

메모리 관리, 상태 유지, 행동 선택, 도구 사용 결과 처리를 총괄합니다.

Think → Act → Observe → Repeat

• 추론 프레임워크

ReAct(Reasoning+Acting), CoT(Chain of Thought) 등의 기법을 통해 논리적인 작업 순서를 계획합니다.



Tools

외부 기능/시스템 호출 인터페이스

인터페이스 역할

언어 모델이 직접 실행할 수 없는 외부 기능이나 시스템을 호출하고 결과를 받아옵니다.

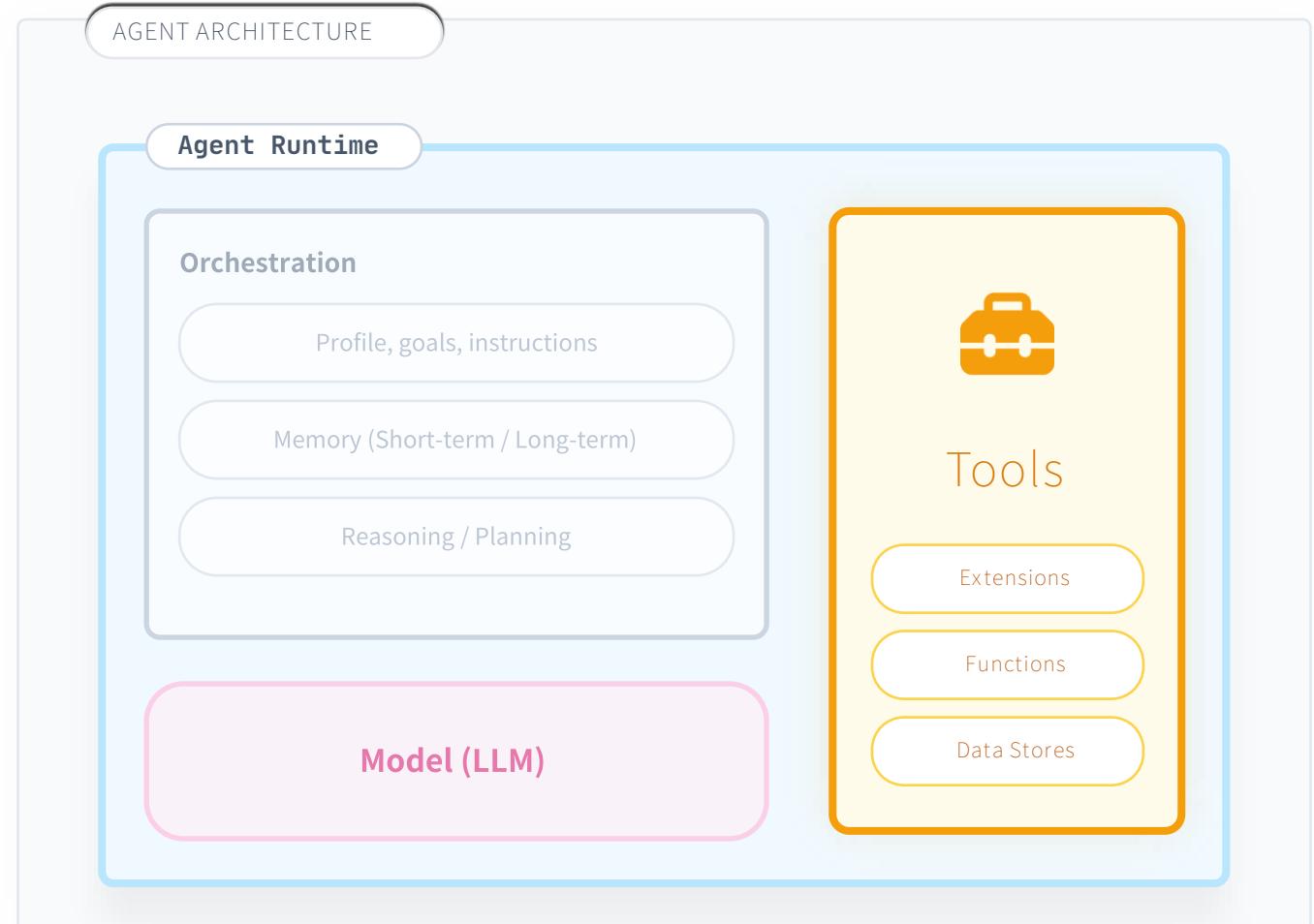
주요 기능

- 최신 정보 검색 (항공편, 날씨, 주가 등)
- 외부 시스템 상호작용 (이메일, DB 등)
- 계산, 변환, 포맷 정리 등 복잡한 작업
- 사실 기반 응답 생성 보완

도구 유형 (Types)

Extensions Functions Data Stores

에이전트가 외부 세계와 상호작용하는 방식을 정의합니다.



TOOLS: Extensions

외부 시스템과 소통하는 표준 인터페이스

연결의 가교 (Bridge)

Extension은 에이전트가 외부 API를 직접 호출할 수 없을 때, 이를 표준화된 방식으로 연결해주는 어댑터 역할을 합니다.

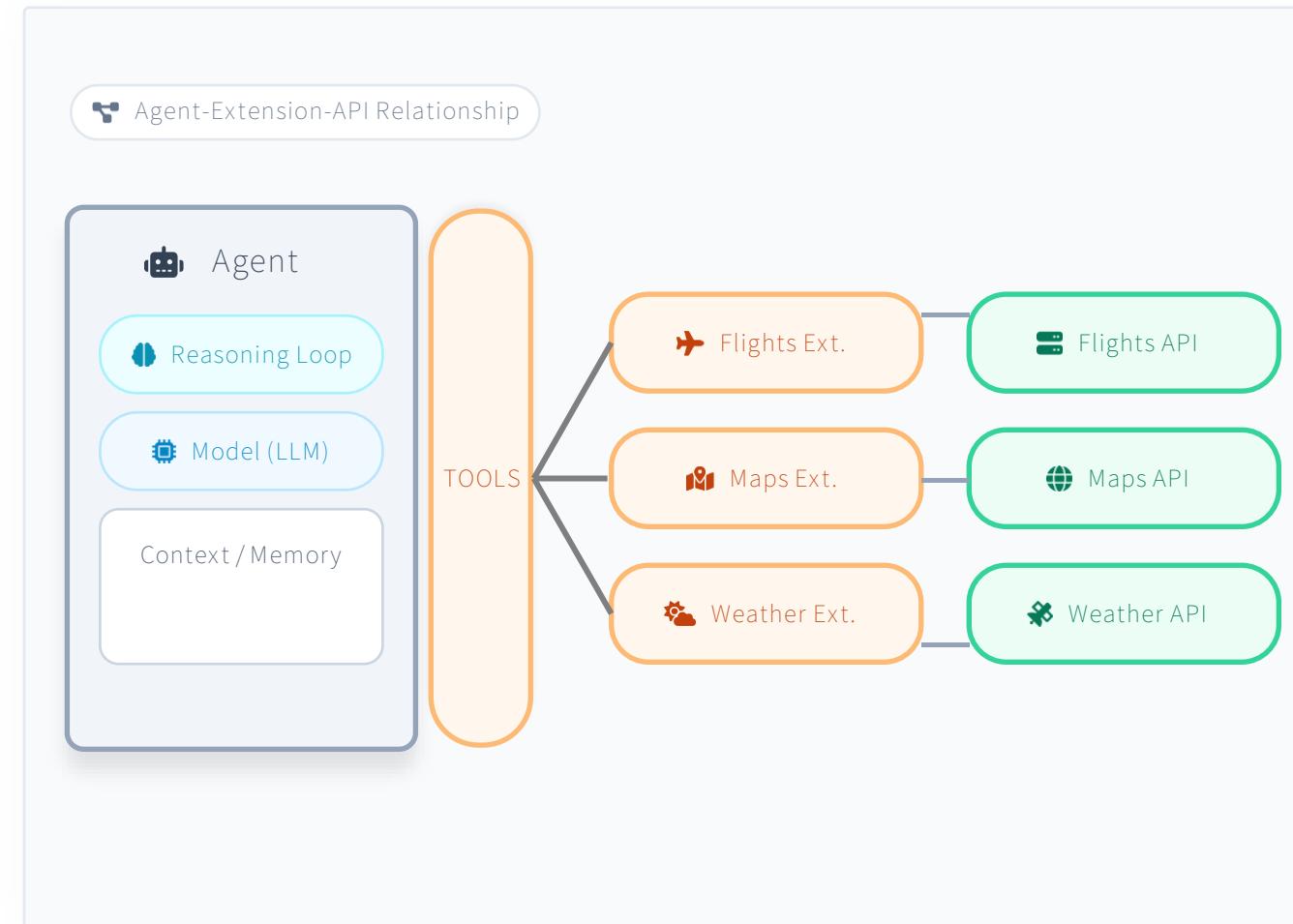
🔌 표준화된 연결 (Interface)

복잡한 외부 API의 명세를 에이전트가 이해할 수 있는 형식으로 단순화하여, 성공적인 호출을 보장합니다.

📖 사용 설명서 (Instruction)

API 엔드포인트 사용법, 필수 인자(Arguments), 그리고 예시 (Examples)를 제공하여 에이전트에게 사용 방법을 학습시킵니다.

하나의 에이전트는 여러 개의 Extension을 동시에 가질 수 있어, 날씨, 지도, 예약 등 다양한 기능을 동시에 수행할 수 있습니다.



TOOLS: Functions

제어권의 이동: Client-Side Execution

유연한 기능 확장

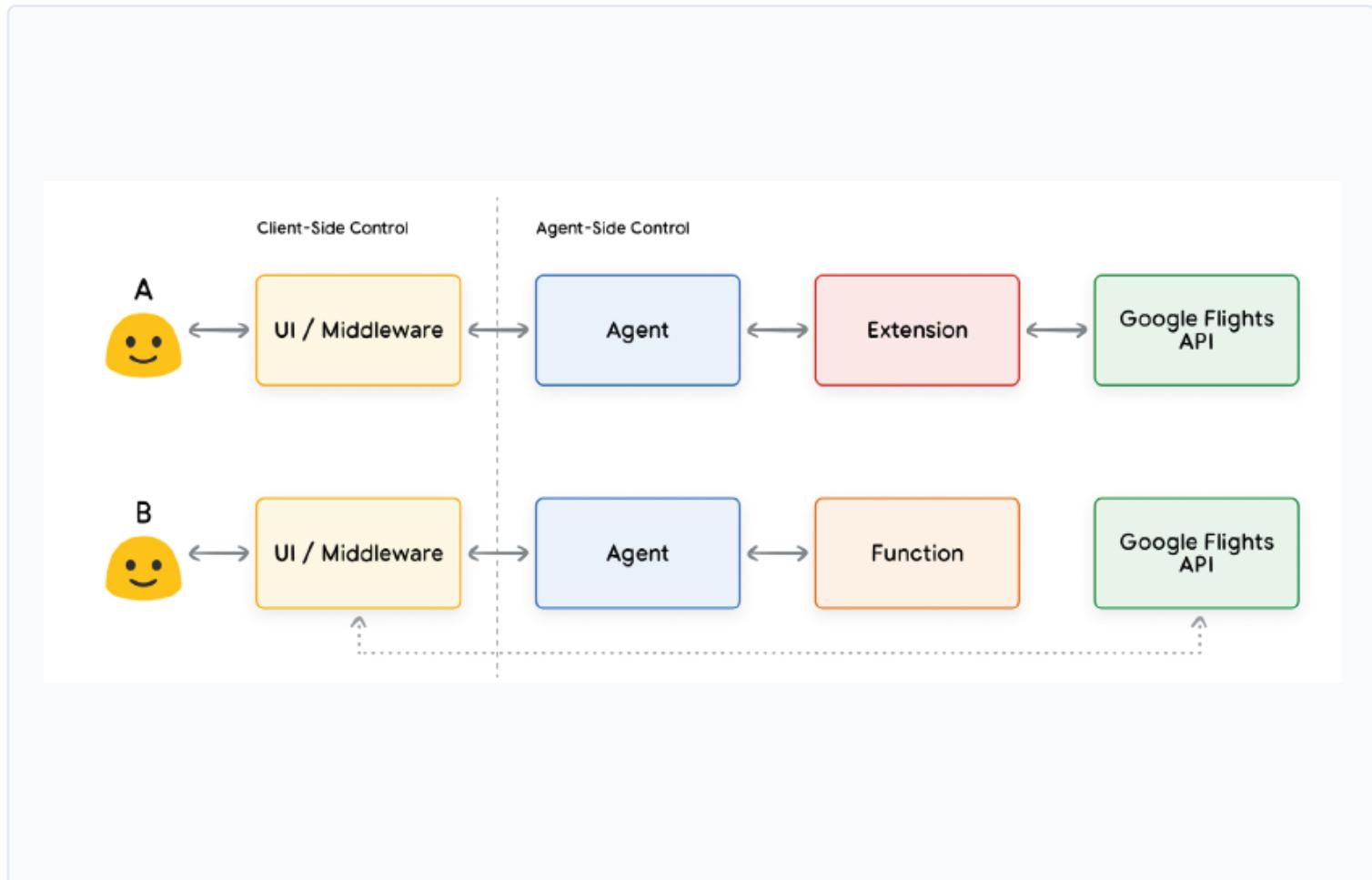
에이전트가 직접 실행할 수 없는 코드를 함수 호출 (Function Call) 형태로 클라이언트에게 요청하여 실행합니다.

</> 작동 원리

SW 엔지니어링의 함수와 유사하게 작동합니다. 모델은 함수명과 인자(Arguments)를 생성할 뿐, 실제 API 호출은 수행하지 않습니다.

☒ 실행 제어권 (Client-Side)

Function은 클라이언트(앱/웹) 측에서 코드가 실행되므로, 개발자가 데이터 흐름과 보안에 대해 훨씬 더 높은 제어권을 가집니다.



TOOLS: Data Stores

외부 지식과 메모리의 확장

지식의 한계 극복

모델의 지식은 학습 데이터에 제한되지만, Data Stores는 이러한 한계를 해결하여 더욱 동적이고 최신 정보에 접근할 수 있도록 합니다. 모델의 응답이 사실에 기반하고 관련성이 유지되도록 보장합니다.

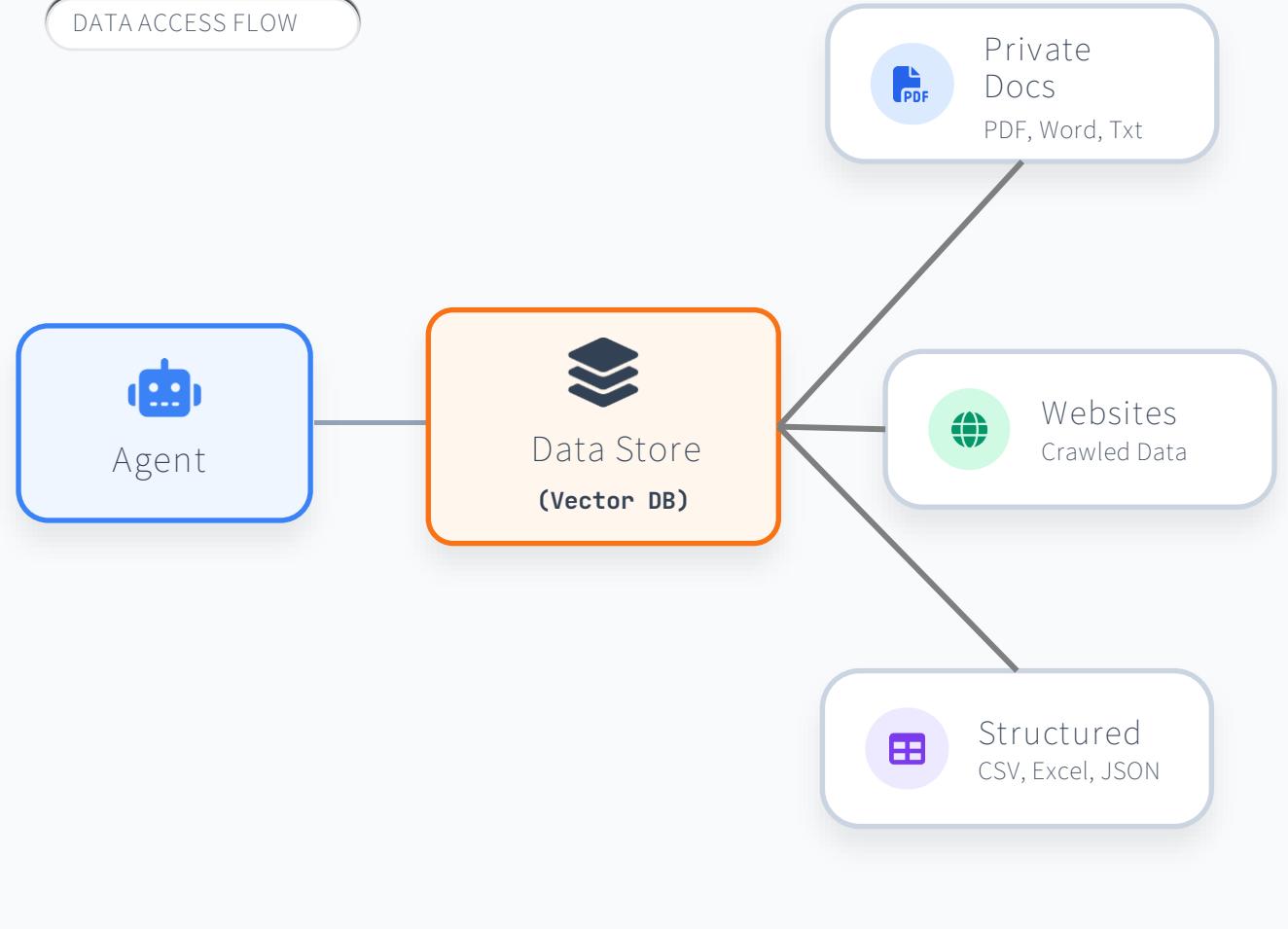
다양한 데이터 형식 지원

개발자는 스프레드시트, PDF, 웹사이트 등 다양한 추가 데이터를 에이전트에게 원래 형식 그대로 제공할 수 있습니다.

벡터 데이터베이스 구현

Data Stores는 일반적으로 실행 시 에이전트가 접근할 수 있는 벡터 데이터베이스로 구현되어, 효율적인 정보 검색과 추출을 지원합니다.

DATA ACCESS FLOW

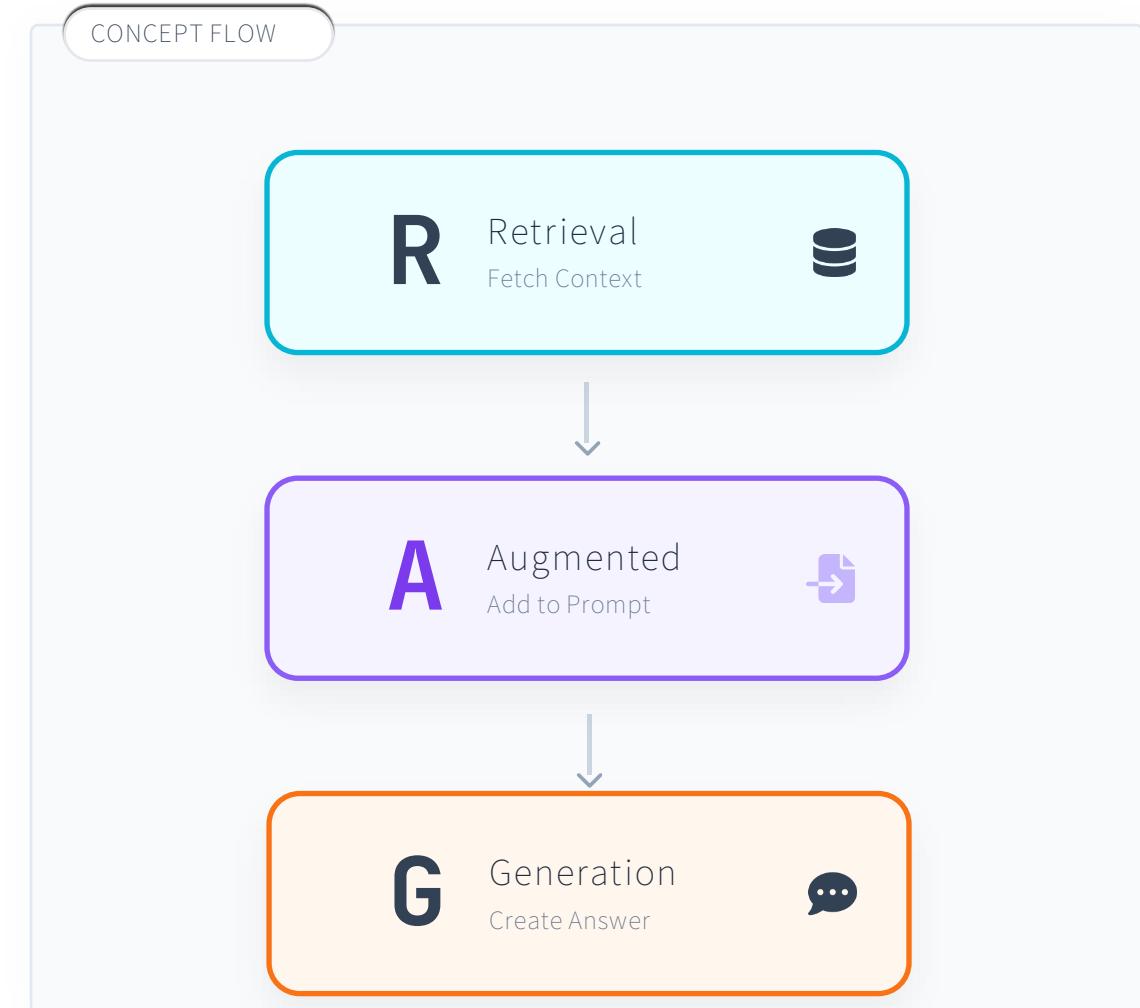


RAG (Retrieval-Augmented Generation)

데이터 스토어의 가장 대표적인 활용 사례

RAG 프로세스 핵심 요소

LLM의 지식을 외부 데이터로 확장하여 할루시네이션을 줄이고 최신 정보를 반영하는 기법입니다.

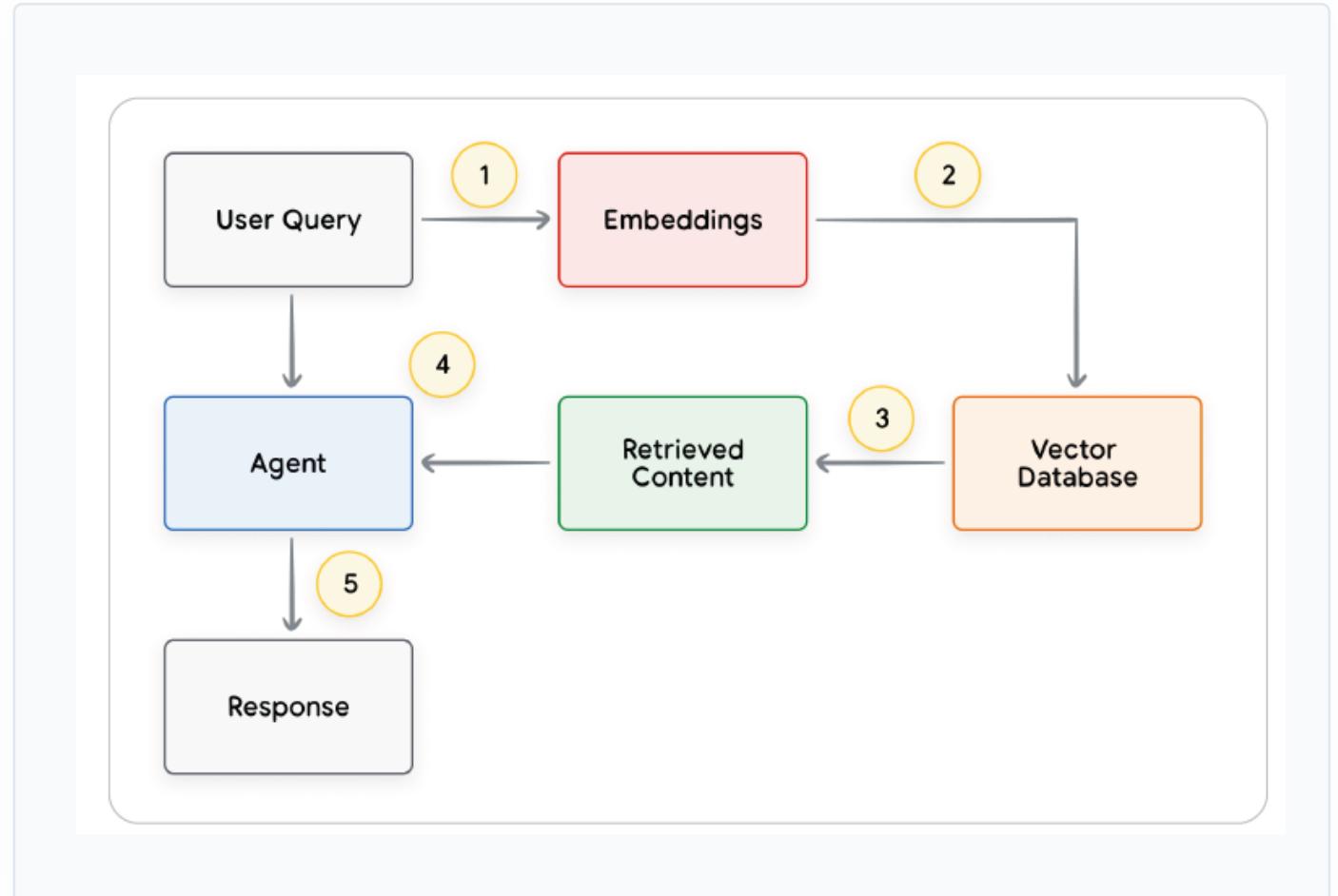


RAG 프로세스 상세

Retrieval Augmented Generation

5단계 순환 프로세스

1. Query Embedding 사용자 쿼리가 임베딩 모델에 전송되어 의미론적 벡터로 변환됩니다.
2. Vector Matching 쿼리 임베딩을 벡터 데이터베이스와 대조하여 의미적으로 유사한 내용을 매칭합니다.
3. Content Retrieval 매칭된 컨텐츠를 텍스트 형식으로 검색(Retrieval)하여 가져옵니다.
4. Context Injection 에이전트에 검색된 컨텐츠를 참고 자료(Context)로 전달하여 응답을 생성합니다.
5. Response Delivery 최종 생성된 응답을 사용자에게 전달합니다.



프롬프트 엔지니어링의 핵심 요소

4 Pillars of a Perfect Prompt

좋은 프롬프트는 AI에게 명확한 가이드를 제공하기 위해 보통 다음의 네 가지 핵심 요소를 포함합니다.



지시 (Instruction)

AI가 수행해야 할 구체적인 작업이나 명령을 정의합니다.

예: "요약해줘", "번역해줘"



맥락 (Context)

작업의 배경지식, 페르소나, 또는 상황을 설정합니다.

예: "너는 10년 차 마케팅 전문가야"



입력 데이터 (Input Data)

처리가 필요한 구체적인 정보나 텍스트를 제공합니다.

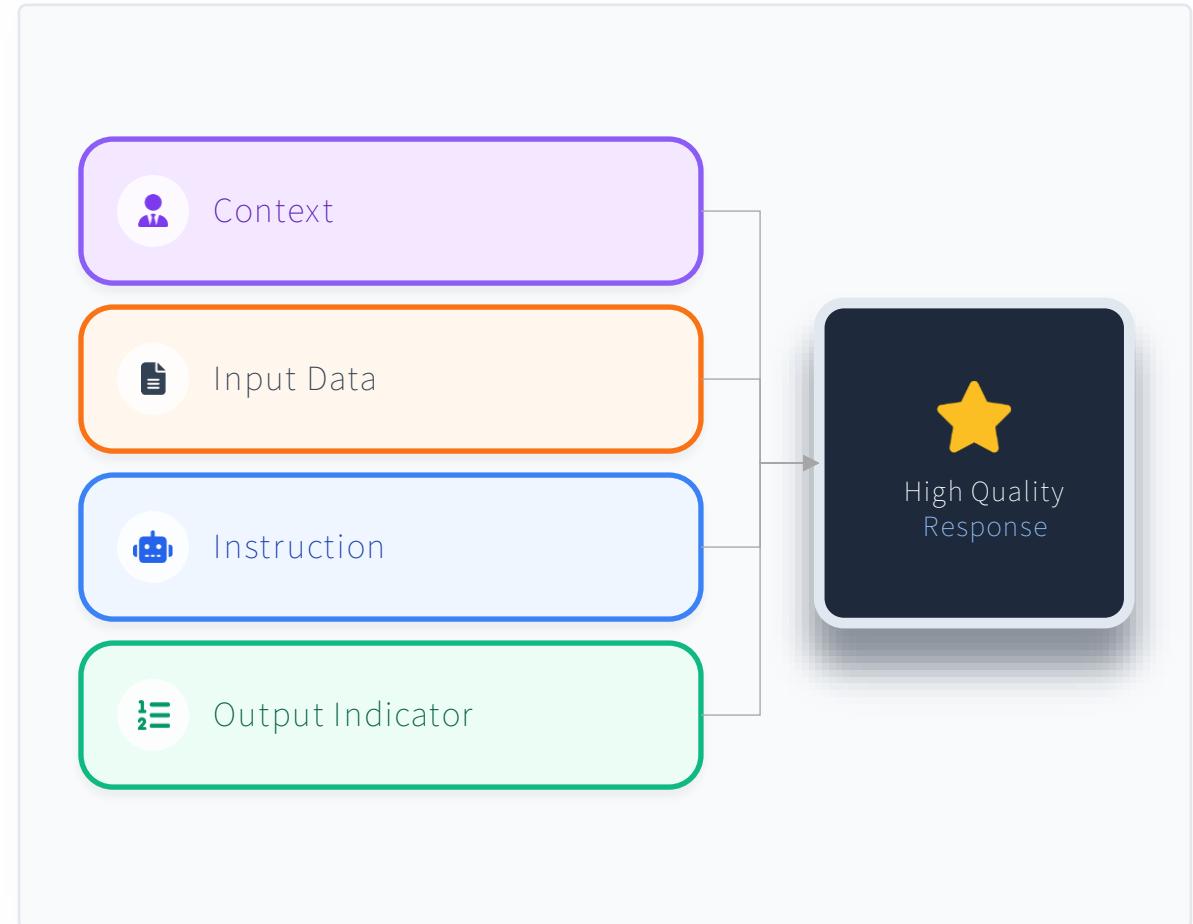
예: [뉴스 기사 전문]



출력 형식 (Output Indicator)

원하는 답변의 형태, 길이, 스타일을 지정합니다.

예: "표 형식으로 정리해줘", "3줄 이내"



나쁜 프롬프트 VS 좋은 프롬프트

AI는 구체적일수록 더 뛰어난 성능을 발휘합니다. 아래 비교를 통해 차이점을 확인해보세요.

예시 1: 마케팅 문구 작성 CREATIVE



BAD PROMPT

"운동화 광고 문구 하나 써줘."

→ 결과: "최고의 착용감, 지금 바로 만나보세요!"와 같은 아주 뻔하고 평범한 문구가 나옵니다.



GOOD PROMPT

"너는 20대 여성들을 타겟으로 하는 친환경 운동화 브랜드의 카피라이터야. 이번에 출시된 '에코런' 운동화는 재활용 플라스틱으로 만들었지만 디자인이 매우 세련된 것이 특징이야. 인스타그램 게시물에 올릴 감성적이고 짧은 문구 3개를 해시태그와 함께 추천해줘."

예시 2: 정보 요약 ANALYTICAL



BAD PROMPT

"이 글 요약해줘. [본문 내용]"

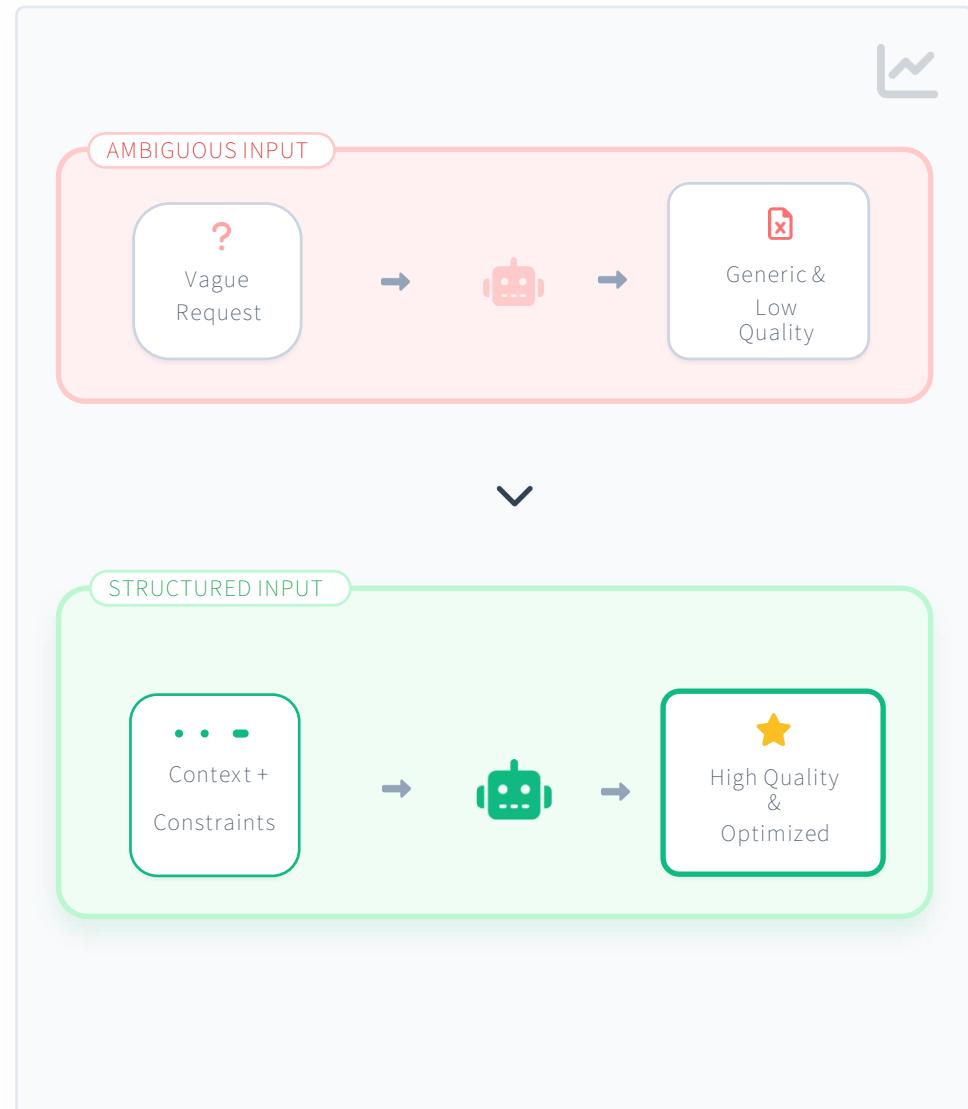
→ 결과: 중요한 내용이 빠지거나, 사용자가 원치 않는 너무 긴 요약이 나올 수 있습니다.



GOOD PROMPT

"아래 제공된 뉴스 기사를 초등학생이 이해할 수 있는 수준으로 쉽게 요약해줘. 불렛 포인트(•) 형식을 사용하고, 전체 길이는 3문장 이내로 작성해줘. [본문 내용]"

구체성(Specificity)이 성능의 차이를 만듭니다



03

SECTION

SETUP

ENVIRONMENT

개발 환경 구축하기

AI 에이전트 개발을 시작하기 위한 필수 도구 설치와 API 연동 준비를 진행합니다.

KEY REQUIREMENTS

- Python 설치 및 가상환경 구성
- Cursor IDE 설치 및 설정
- 프로젝트 폴더 구조 생성
- OpenAI API 키 발급 및 설정

Python 설치 및 환경 설정



파이썬 설치

python.org에서 3.11+ 버전을 다운로드합니다. Add to PATH 체크 필수.

[Download from python.org](#)

01



설치 확인

터미널을 열고 명령어를 입력하여 정상 설치를 확인합니다.

\$python --version

02



프로젝트 폴더 생성

작업할 디렉토리를 생성하고 해당 폴더로 이동합니다.

생성 `mkdir ai_research_assistant`

이동 `cd ai_research_assistant`

03



가상환경 생성

프로젝트 폴더 내에 독립된 가상환경(venv)을 생성합니다.

`$python -m venv venv`

04



가상환경 활성화

운영체제에 맞는 실행 명령어를 입력합니다.

Win > `venv\Scripts\activate`

Mac \$ `source venv/bin/activate`

05



활성화 확인

프롬프트 앞의 (venv) 표시를 확인합니다.

`(venv)user@pc:~$`

06

Cursor IDE 소개

기존의 VS Code를 포크(Fork)하여, 강력한 AI 기능을 개발 환경 자체에 깊숙이 통합한 차세대 에디터입니다.

AI 네이티브 기능

Core

단순 플러그인이 아닌 IDE 자체가 AI를 위해 설계되었습니다.

- Cursor Tab: 개발자 의도를 파악해 여러 줄 코드 제안
- Composer Ctrl+I: 다중 파일 동시 수정 및 생성

익숙한 사용 환경

VS Code 기반으로 만들어져 100% 호환성을 보장합니다. 익숙한 단축키, 테마, 확장 프로그램을 그대로 사용하며 클릭 한 번으로 설정을 가져올 수 있습니다.

인텔리전트 대화 & 수정

- Chat Ctrl+L: @Codebase 등 맥락 기반 질문
- Edit Ctrl+K: 자연어로 실시간 코드 수정 명령

KEY FEATURES

AI Native
Tab & ComposerIndexing
Codebase RAGCursor
! LI59Smart Chat
Context AwareVS Code
Full Compatibility

Cursor IDE 설치 및 설정



01 공식 사이트 접속

웹 브라우저를 열고 Cursor 공식 홈페이지에 접속합니다.

<https://cursor.com>

01



02 설치 파일 다운로드

사용 중인 OS(Windows, Mac)에 맞는 버전을 다운로드합니다.

Mac / Windows

02



03 PATH 추가

터미널 실행을 위해 설치 시 'Add to PATH' 옵션을 반드시 체크하세요.

Add to PATH (Required)

03



04 회원가입 및 로그인

설치 후 실행하여 계정을 생성하거나 로그인합니다.

Github G Google

04



05 Cursor Pro 적용

강력한 AI 기능을 위해 Pro 플랜(7일/14일 무료)을 활성화합니다.

Trial Active(7 Days Free)

05



06 VS Code 설정 가져오기

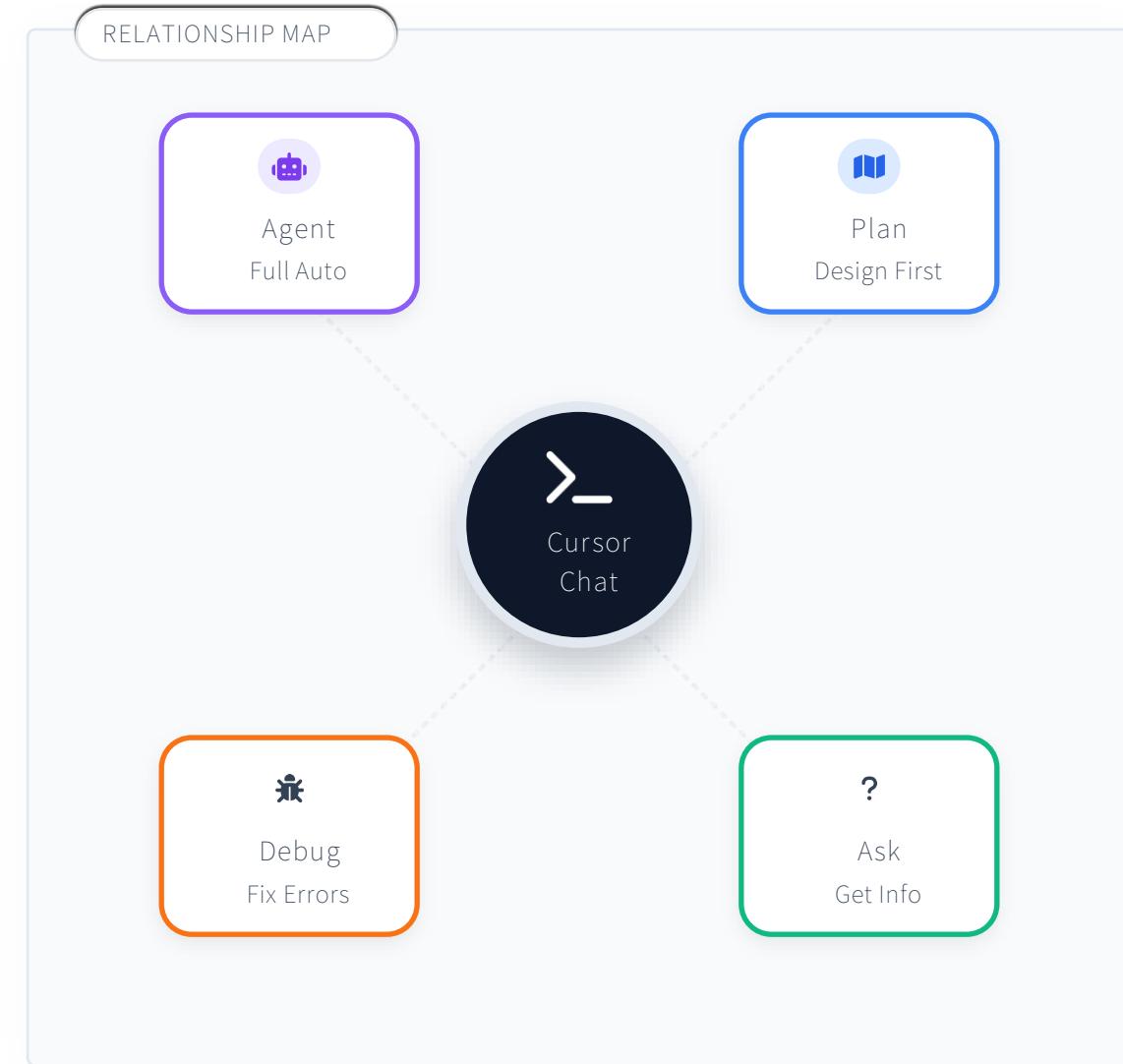
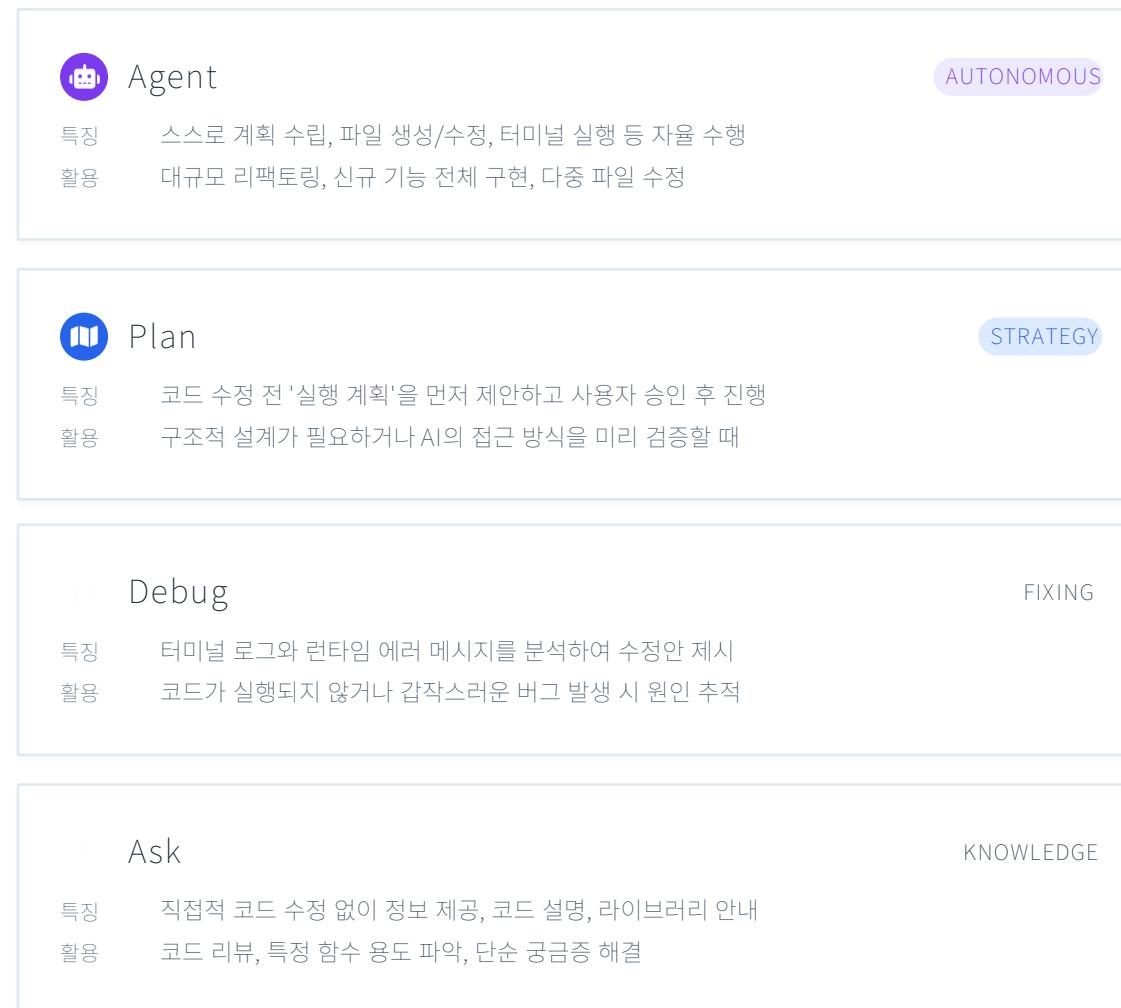
기존 VS Code의 확장 프로그램과 테마를 한 번에 가져옵니다.

→ One-click Import

06

Cursor 4가지 채팅 모드

상황에 맞는 최적의 모드 선택



OpenAI 계정 생성

시작하기 전 준비사항

API 키를 발급받기 위해서는 계정 생성 후 결제 수단 등록이 필수적입니다.

1 Step 1: 회원가입

OpenAI 플랫폼에 접속하여 이메일 인증을 완료하고 계정을 생성합니다.

2 Step 2: 결제 정보 등록

Settings > Billing 메뉴에서 신용카드 또는 직불카드를 등록합니다.

3 Step 3: 크레딧 충전

API 호출을 위해 최소 \$5 이상의 크레딧을 충전합니다. (Pre-paid)

The screenshot shows the OpenAI Platform interface. On the left, a sidebar menu includes 'Get started', 'Overview' (which is selected), 'Quickstart', 'Models', 'Pricing', 'Libraries', 'Docs MCP', 'Latest: GPT-5.2', 'Core concepts' (with 'Text generation', 'Code generation', 'Images and vision', 'Cookbook', and 'Forum' listed), and 'Billing' (which is highlighted with a red circle labeled '2'). The main content area has a heading 'Developer quickstart' and a code snippet in JavaScript:

```

javascript ◊
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const response = await client.responses.create({
5   model: "gpt-5.2",
6   input: "Write a short bedtime story about a unicorn.",
7 });
8
9 console.log(response.output.text);

```

Below this is a 'Billing' section with tabs for 'Overview' (selected), 'Payment methods', 'Billing history', 'Credit grants', and 'Preferences'. The 'Overview' tab displays a credit balance of '\$9.12' and buttons for 'Add to credit balance' and 'Cancel plan'. A note says 'Auto recharge is off' and provides instructions to enable it. Other tabs include 'Payment methods' (with a sub-note to add or change payment method) and 'Billing history' (with a sub-note to view past and current invoices).

API 키 발급

OpenAI 서비스 연동을 위한 인증 키 생성

1

API Keys 페이지 이동

platform.openai.com/api-keys로 접속하여 로그인합니다.

2

새 키 생성 (Create new secret key)

우측 상단의 버튼을 클릭하고 키의 용도(예: "ai-research-assistant")를 이름으로 지정합니다.

3

키 복사 및 저장 IMPORTANT

생성된 키(sk-...)를 복사하여 안전한 곳에 저장합니다.

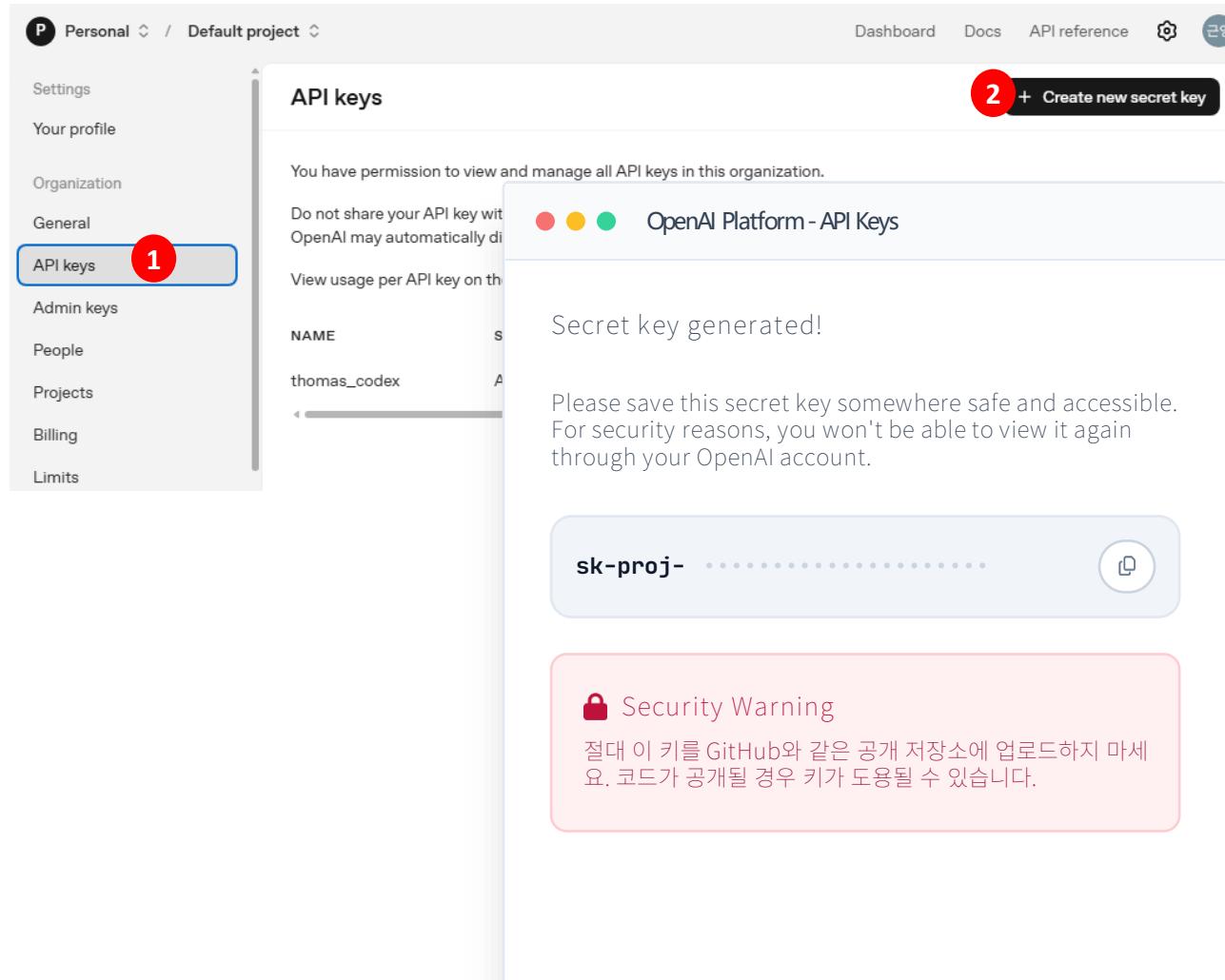
주의: 창을 닫으면 다시는 키를 확인할 수 없습니다.

분실 시 새로 발급받아야 합니다.

4

권한 설정 (Permissions)

프로젝트 요구사항에 따라 권한을 설정합니다. (기본: All 또는 Read/Write 권한 부여)



프로젝트 폴더 구조

구조화된 개발 환경

코드를 기능별로 분리하고 확장성을 고려하여 폴더를 구성합니다.

</> **src/** (Source)

에이전트의 핵심 로직이 위치합니다. 대화 관리자, 에이전트 모듈, 도구 함수 등이 포함됩니다.

data/ (Storage)

벡터 DB(ChromaDB) 데이터 파일과 에이전트가 생성한 리포트 결과물이 저장됩니다.

config/ (Settings)

LLM 프롬프트 템플릿, 모델 설정, 상수 값 등 변경 가능한 설정 정보들을 관리합니다.

PROJECT EXPLORER

```
ai-research-assistant
├── venv/ # Virtual Environment
└── src/
    ├── __init__.py
    ├── conversation_manager.py
    └── search_agent.py
└── data/
    └── chroma_db/ # Vector DB
└── config/
    └── prompts.py
├── .env
└── requirements.txt
└── main.py
└── README.md
```

효율적인 개발을 위한 표준 디렉토리 구조

프로젝트 스켈레톤 생성 프롬프트

AI 리서치 어시스턴트 프로젝트를 시작하려고 합니다.

다음 요구사항에 맞는 프로젝트 폴더 구조를 생성해주세요:

[프로젝트 구조]

```
ai-research-assistant/
├── src/
│   ├── __init__.py
│   └── conversation_manager.py
├── data/
└── config/
    └── prompts.py
├── .env
├── .gitignore
├── requirements.txt
└── main.py
└── README.md
```

[요구사항]

1. 모든 폴더와 파일을 생성해주세요
2. `.gitignore` 파일에는 다음을 포함:

- `.env`
- `venv/`
- `__pycache__/`
- `*.pyc`
- `.DS_Store`

3. `requirements.txt`에는 다음 패키지 포함:

- `openai==1.12.0`
- `python-dotenv==1.0.0`

4. `README.md`에는 프로젝트 설명과 설치 방법 작성

5. `.env` 파일에는 `OPENAI_API_KEY` 템플릿 추가 (실제 키는 나중에 입력)

환경변수 설정

보안 설정의 핵심

API 키는 코드와 분리하여 관리해야 하며, 절대 Git 저장소에 업로드되어서는 안 됩니다.

1. .env 파일 생성

프로젝트 루트에 생성하며, KEY=VALUE 형식으로 비밀 키를 저장합니다.

2. .gitignore 설정

.env를 예외 목록에 추가하여 GitHub 등에 실수로 업로드되는 것을 방지합니다.

3. Python에서 사용

python-dotenv 라이브러리를 통해 환경변수를 안전하게 로드합니다.



Don't commit .env file!



OPENAI_API_KEY=
sk-proj-1234xxxx...



main.py

```
from dotenv import load_dotenv  
import os  
  
# 환경변수 로드  
load_dotenv()  
  
api_key =  
os.getenv("OPENAI_API_KEY")
```

필수 라이브러리 설치

Python 패키지 구성 및 설치

핵심 패키지 개요

1주차 프로젝트 진행을 위해 OpenAI API 연동과 환경변수 관리 라이브러리가 필요합니다.

openai

OpenAI의 GPT 모델과 통신하기 위한 공식 Python 클라이언트 라이브러리입니다. (v1.12.0+)

python-dotenv

API 키와 같은 민감한 정보를 .env 파일에서 안전하게 로드하여 관리합니다.

UPCOMING LIBRARIES

tavily-python (Week 2)

chromadb (Week 3)

langchain (Week 4)

requirements.txt

```
# AI Agent Core Dependencies
openai==2.15.0
python-dotenv ==1.0.0
# Optional for debugging
colorama ≥ 0.4.6
```

Terminal - Install

```
→ai-project git:(main) pip install -r requirements.txt
Collecting openai==1.12.0
  Downloading openai-1.12.0-py3-none-any.whl (226 kB)
Collecting python-dotenv==1.0.0
  Downloading python_dotenv-1.0.0-py3-none-any.whl (19 kB)
Installing collected packages: python-dotenv, openai
Successfully installed openai-1.12.0 python-dotenv-1.0.0
```

설치 확인 및 테스트

연결 테스트 절차

아래 3단계를 통해 OpenAI API 호출이 정상적으로 이루어지는지 검증합니다.

1 테스트 파일 생성

test_connection.py 파일을 프로젝트 루트에 생성하고 오른쪽 코드를 작성합니다.

2 코드 실행

터미널에서 가상환경이 활성화된 상태로 스크립트를 실행합니다.

```
$ python test_connection.py
```

3 결과 확인

AI가 "Hello!"에 대한 응답을 출력하면 연동 성공입니다. 오류 발생 시 에러 메시지를 확인하세요.

API 연동이 정상적으로 동작하는지 확인합니다.

test_connection.py

```
from openai import OpenAI
from dotenv import load_dotenv
import os

# 환경변수 로드 (.env)
load_dotenv()

# 클라이언트 초기화
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# API 호출 테스트
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "Hello!"}]

    print(response.choices[0].message.content)
```

```
user@project:~$ python test_connection.py
```

환경 구축 체크리스트

PRE-REQUISITES

이 단계가 완료되지 않으면 다음 실습을 진행할 수 없습니다.

- Python 3.11+ 설치 완료



SYSTEM

- 가상환경 활성화 및 라이브러리 설치

LIBS

- OpenAI 계정 / 결제 / 키 발급

API

- .env / .gitignore 설정 완료

CONFIG

- API 연결 테스트 성공

TEST

실습을 시작하기 전 필수 확인 사항입니다.



Setup Complete

이제 AI 에이전트 개발을 시작할 준비가 되었습니다.

5
ITEMS
100%
READY

TROUBLESHOOTING

문제 해결 가이드



인증 실패 (Authentication)

API 키가 올바른지, 환경변수가 로드되었는지 확인하세요.

ERR 401



요청 한도 초과 (Rate Limit)

요청 간격을 조정하고 지수 백오프(Exponential Backoff)를 적용하세요.

ERR 429



네트워크 오류 (Network)

방화벽, 프록시 설정 및 타임아웃 시간을 점검하세요.

CONNECTION



토큰 초과 (Context Length)

max_tokens를 줄이거나 대화 히스토리(Memory)를 요약하세요.

TOKEN



라이브러리 호환성 (SDK)

DEPENDENCY



Debug & Fix

정확한 오류 코드를 이해하면 문제의 90%는 해결됩니다.

401

429

5xx

AUTH

RATE

SERVER

04

SECTION

HANDS-ON

BASIC SYSTEM

실습 Part 1 기본 대화 시스템

OpenAI API를 연동하여 대화 히스토리를 관리하고, 실제 동작하는 CLI 기반 챗봇을 구현합니다.

ACTIVITY CHECKLIST

40 MIN

▶ OpenAI API 연동

▶ 대화 히스토리 관리

>_ 입출력 인터페이스 구현

▶ 코드 실행 및 테스트

실습 목표

기본 대화 시스템 구축을 위한 4가지 달성 과제



01 OpenAI API 연동

Python 환경에서 openai 라이브러리를 사용하여 API 키 인증과 클라이언트 초기화를 수행합니다.

01



02 사용자 입력 수집

터미널 환경(CLI)에서 사용자의 자연어 입력을 실시간으로 받고, 이를 API가 이해할 수 있는 메시지 포맷으로 변환합니다.

02



03 대화 루프 구현

일회성 응답이 아닌, 연속적인 대화가 가능하도록 while 루프와 히스토리 관리(Context) 로직을 구축합니다.

03



04 예외 처리 확인

API 연결 오류, 토큰 초과 등 발생 가능한 문제를 사전에 차단하고 안정적인 실행 환경을 보장하는 코드를 작성합니다.

04

실습 Part1 프로젝트 구조

기본 대화 시스템 폴더 구성

주요 디렉토리 설명

Python 프로젝트의 표준 구조를 따르며, 설정, 데이터, 코드를 명확히 분리합니다.

src/

소스 코드가 위치하는 핵심 디렉토리입니다.
conversation_manager.py가 대화 로직의 중심 역할을 수행합니다.

config/

프로젝트 설정을 관리합니다. AI 페르소나와 시스템 프롬프트 템플릿은
모두 이곳의 prompts.py에서 정의합니다.

Environment

.env 파일로 API 키를 안전하게 관리하고, venv/를 통해 프로젝트 전용
가상환경을 격리합니다.

```
ai-research-assistant
  ↘ AI-research-assistant/
    └── venv/ # 가상 환경 디렉토리
    └── src/
      ├── __init__.py # 패키지 초기화 파일
      ├── conversation_manager.py # 대화 관리 핵심 모듈
      └── test_connection.py # API 연결 테스트 스크립트
    └── config/
      ├── config/
        └── prompts.py # 프롬프트 템플릿 설정
      └── data/ # 데이터 저장 디렉토리 (현재 비어있음)
    └── .env # 환경 변수 (Git 제외)
    └── .gitignore # Git 무시 파일 목록
    └── requirements.txt # Python 패키지 의존성
    └── main.py # 매인 실행 파일
    └── README.md # 프로젝트 설명서
    └── IMPROVEMENTS.md # 개선 사항 문서
    └── conversation.log # 로그 파일 (실행 시 생성)
```

프로그램 실행 흐름

실행 단계 (Execution Steps)

Python 스크립트 실행부터 종료까지의 전체 프로세스입니다.

1 초기화 (Initialization)

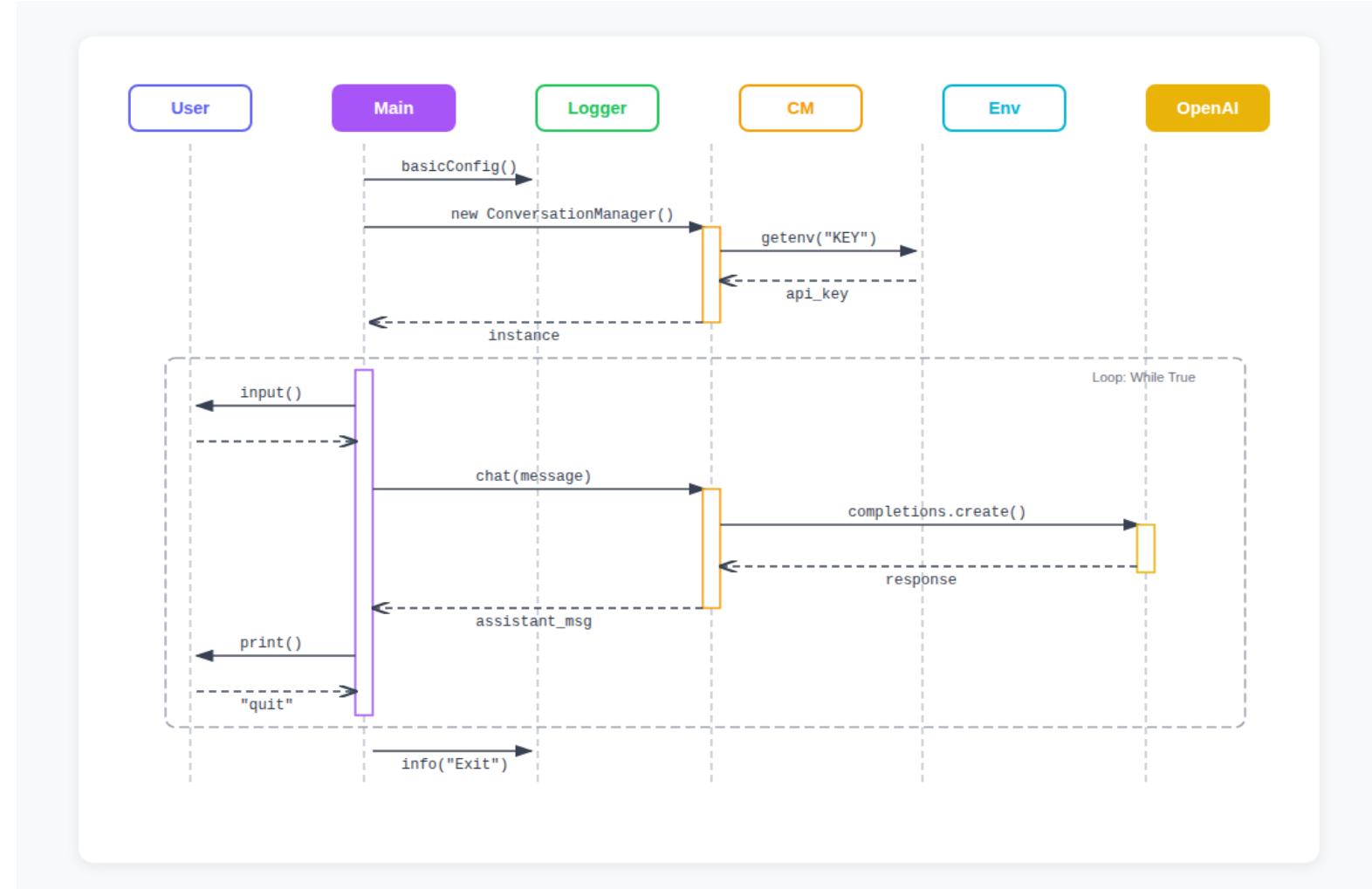
main() 시작 시 로깅을 설정하고, ConversationManager를 인스턴스화합니다. 이 과정에서 .env 로드와 OpenAI 클라이언트 연결이 수행됩니다.

2 대화 루프 (Chat Loop)

while True: 루프 안에서 사용자 입력을 받고, chat() 메서드를 통해 AI 응답을 생성하여 출력합니다.

3 오류 처리 및 종료 (Termination)

API 오류나 네트워크 문제를 로깅하고, 사용자가 'quit' 입력 시 루프를 종료하며 총 대화 횟수를 리포트합니다.



main.py 소스 구조 설명

Python 표준 라이브러리 활용

```
import logging
import sys
```

✓ Python 표준 라이브러리 - 설치 없이 바로 사용 가능

.logging module

이벤트 로깅 시스템으로 디버깅과 상태 추적을 담당합니다.
주요 기능: 로그 레벨 설정, 파일/콘솔 출력, 로그 포멧팅

로그레벨 종류:

DEBUG INFO WARNING ERROR CRITICAL

> sys module

시스템 관련 파라미터와 함수를 제공하는 인터페이스입니다.

주요 기능:

`sys.stdout/stderr`

표준 출력/에러 스트림

`sys.exit()`

프로그램 종료

`sys.argv`

명령줄 인자

`sys.path`

모듈 검색 경로

LIBRARY ECOSYSTEM COMPARISON

표준 라이브러리

Built-in (설치 불필요)

os

sys

logging

json

datetime

re

collections

외부 패키지

pip install 필요

requests

numpy

pandas

main.py 소스 구조 설명 (이어서)

로깅 설정 및 초기화

```
# 로깅 설정
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('conversation.log', encoding='utf-8'),
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)
```

.logging.basicConfig() – 로깅 기본 설정

level logging.INFO: INFO 레벨 이상의 로그만 기록 (DEBUG 제외)

format 로그 메시지의 출력 형식 지정

%(asctime)s: 시간

%(name)s: 로거 이름

%(levelname)s: 로그 레벨

%(message)s: 메시지

handlers 로그가 출력될 대상 지정

FileHandler: 파일(conversation.log)에 기록

StreamHandler: 터미널 콘솔에 실시간 출력

logging.getLogger()

`__name__` 현재 실행 중인 모듈의 이름을 나타내는 내장 변수
(예: main)

`getLogger()` 해당 이름을 가진 로거 인스턴스를 반환하여, 로그 출처를 명확히 함

Tip: 로거 이름을 지정하면 어떤 파일에서 로그가 발생했는지 추적하기 쉽습니다.

예시: 간단한 이름
`logger = logging.getLogger("main_logger")`

main.py 소스 구조 설명 - 이어서

메인 함수 & 클래스 초기화

```
def main():
    """메인 함수 - 대화 루프 실행"""
    # 환영 메시지 출력
    print("=" * 60)
    print("AI 어시스턴트에 오신 것을 환영합니다!")
    print("'quit', 'exit', 또는 '종료'를 입력하면 대화를 종료할 수 있습니다.")
    print("=" * 60)
    print()
```

```
system_message = "당신은 도움이 되는 AI 어시스턴트입니다."
conversation_manager =
ConversationManager(system_message=system_message)
logger.info("ConversationManager 초기화 완료")
```

</> 기본 문법 포인트

- def** 함수를 정의할 때 사용하는 키워드입니다.
- main()** 함수의 고유 이름입니다.
- “문자열” * num** 문자열을 해당 숫자만큼 반복합니다. (구분선 생성용)
- print()** 콘솔에 텍스트를 출력하며, 인자가 없으면 빈 줄을 출력합니다.



ConversationManager 초기화 로직

ConversationManager 클래스를 호출하면 `__init__`가 자동 실행됩니다. `system_message="당신은 도움이 되는 AI 어시스턴트입니다."`가 `__init__`의 `system_message` 파라미터로 전달됩니다. ConversationManager에서는 인스턴스 생성 시, OpenAI API 클라이언트를 초기화하고, `system_message`가 있는 경우, 이를 설정합니다.

```
if system_message:
    self.messages.append({
        "role": "system",
        "content": system_message
    })
```

main.py 소스 구조 설명 - 이어서

대화 루프 처리 로직

```

while True:
    try:
        # 사용자 입력 받기
        user_input = input("사용자: ").strip()

        # 종료 명령어 확인 ('quit', 'exit', '종료')
        if user_input.lower() in ['quit', 'exit', '종료']:
            logger.info("사용자가 종료 명령어를 입력했습니다.")
            break

        # 빈 입력은 무시
        if not user_input:
            print("메시지를 입력해주세요.")
            continue

        # ConversationManager.chat() 호출하여 AI 응답 받기
        ai_response = conversation_manager.chat(user_input)

        # AI 응답 출력
        print(f"AI: {ai_response}")
        print() # 가독성을 위한 빈 줄
    
```



루프 제어에 사용된 문법

`while True`

조건이 항상 참인 무한 반복문

`try`

예외가 발생할 수 있는 코드 블록 시작

`input().strip()`

입력 후 앞뒤 공백 자동 제거

`.lower()`

대소문자 구분 없이 소문자로 변환

`in [list]`

리스트 내 포함 여부 확인 (Membership)

`break`

반복문을 즉시 종료하고 탈출

`continue`

남은 코드를 건너뛰고 다음 반복 시작

`conversation_manager.chat(user_input)`

conversation_manager 메서드 호출. 사용자 입력값을 인자로 전달

`print(f"AI: {ai_response}")`

ai_response로 open AI의 API 응답 값을 받아서,
f-string: f"문자열 {변수}"로 변수 삽입하여 출력

main.py 소스 구조 설명 - 이어서

대화 루프 처리 로직

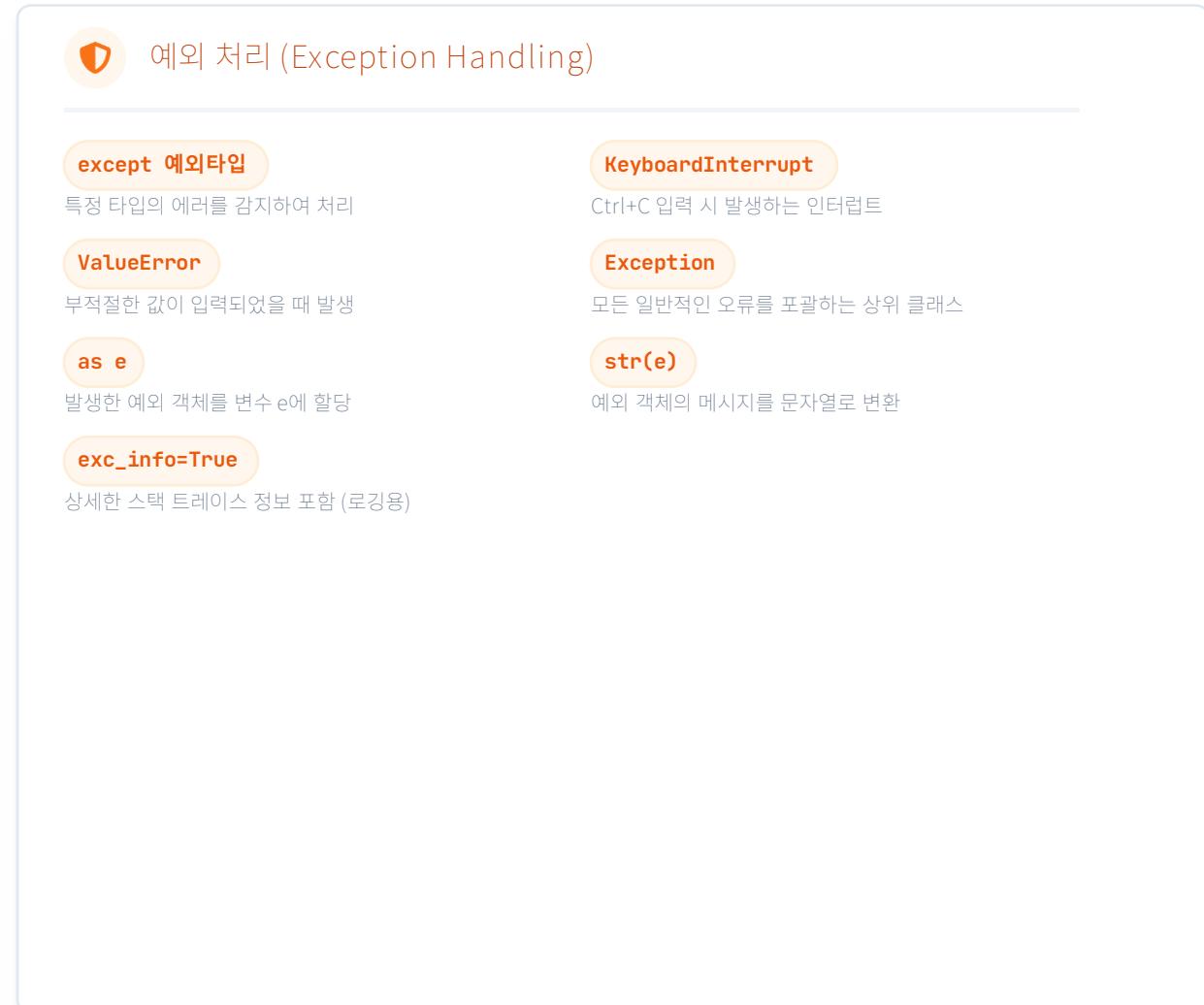
```

except KeyboardInterrupt:
    # Ctrl+C 입력 시 루프 종료
    logger.info("사용자가 Ctrl+C를 눌렀습니다.")
    print("\n") # 줄바꿈
    break

except ValueError as e:
    # 입력 검증 오류 처리
    logger.warning(f"입력 검증 오류: {str(e)}")
    print(f"입력 오류: {str(e)}")
    print()

except Exception as e:
    # 일반 예외 처리
    logger.error(f"오류 발생: {str(e)}", exc_info=True)
    print(f"오류가 발생했습니다: {str(e)}")
    print("계속하려면 Enter를 누르세요...")
    try:
        input()
    except KeyboardInterrupt:
        break
    print()

```



main.py 소스 구조 설명 - 이어서

프로그램 종료 및 정리 로직

```
finally:
    # 깔끔한 종료 메시지 출력
    try:
        message_count = conversation_manager.get_message_count()
        print("=" * 60)
        print("대화를 종료합니다. 안녕히 가세요!")
        print(f"총 대화 횟수: {message_count}회")
        print("=" * 60)
        logger.info(f"프로그램 종료. 총 대화 횟수: {message_count}회")
    except NameError:
        # conversation_manager가 초기화되지 않은 경우
        print("=" * 60)
        print("대화를 종료합니다. 안녕히 가세요!")
        print("=" * 60)
```



finally 블록 문법

finally

예외 발생 여부와 관계없이 항상 실행 되는 코드 블록

Usage

리소스 정리(파일 닫기, DB 연결 해제), 로깅, 종료 메시지 출력에 필수적

Guarantee

try 블록 도중 에러가 나서 프로그램이 멈추더라도, 종료 직전 반드시 실행됨



안전한 종료 처리 (Fallback)

NameError

정의되지 않은 변수에 접근할 때 발생하는 예외

Scenario

프로그램 시작 직후(API 키 오류 등) / 생성 전 종료될 때 발생 가능

Solution

초기화 실패 시에도 사용자에게 깔끔한 종료 메시지를 보여주기 위한 방어적 코딩

대화 히스토리 관리

Stateless API의 한계 극복 전략

왜 히스토리 관리가 필요한가?

LLM API는 Stateless(상태 비저장)이므로, 이전 대화를 기억하지 못합니다. 개발자가 매번 전체 문맥을 제공해야 합니다.

Stateless API Nature

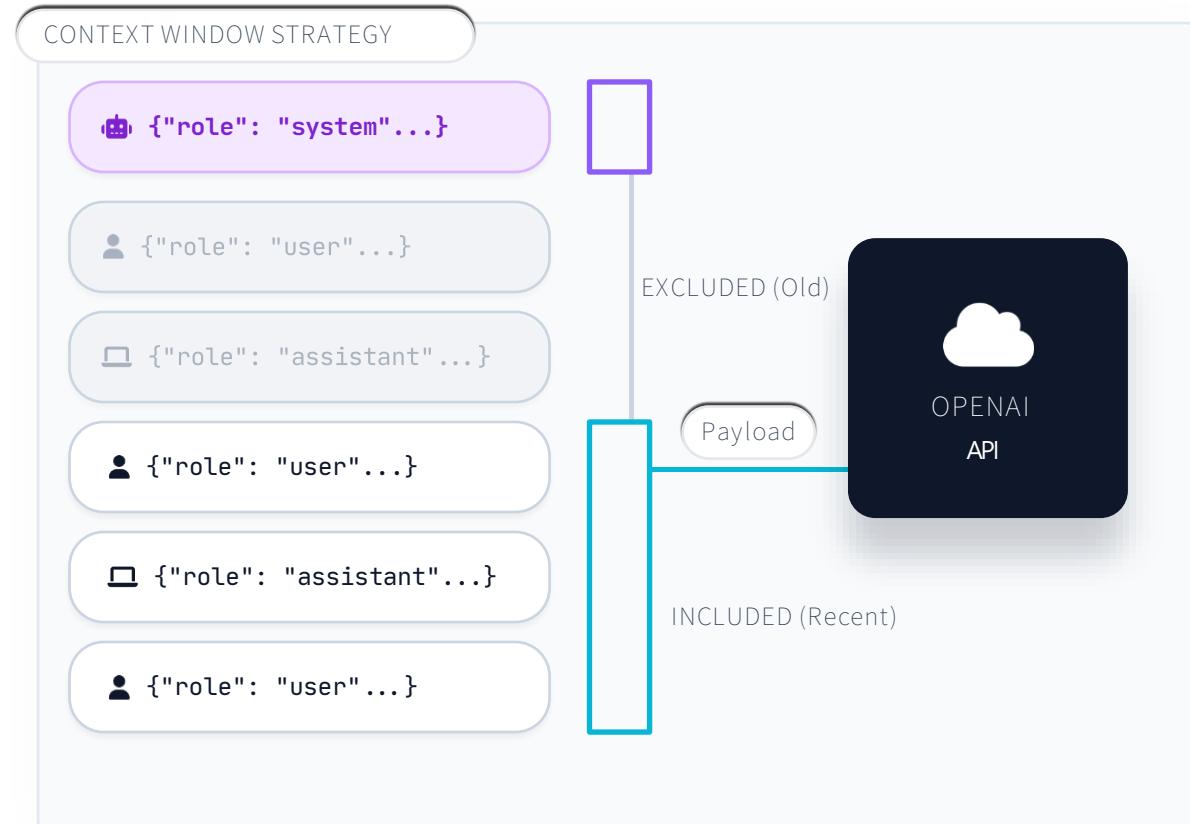
API 요청은 독립적입니다. "그것에 대해 더 말해줘"라고 요청하려면, "그것"이 무엇인지 포함된 이전 대화 전체를 함께 보내야 합니다.

Token Management

무한정 저장할 수 없으므로 Sliding Window 방식(최근 N개 유지)을 사용하여 비용을 절감하고 컨텍스트 제한을 준수합니다.

</> Class-Based Implementation

ConversationManager 클래스를 구현하여 메시지 추가 append), 관리, API 전송로직을 캡슐화하는 것이 효율적입니다.



ConversationManager 클래스 구조

핵심 상태 관리 구조 분석

CLASS STRUCTURE TREE

ConversationManager

ATTRIBUTES (STATE)

`self.client`

OpenAI 클라이언트

`self.messages`

List[MessageDict] - 대화 히스토리

`self.message_count`

int - 대화 횟수

METHODS (BEHAVIOR)

`__init__(system_message)`

초기화

`chat(user_input) -> str`

대화 처리 (핵심)

`get_messages() -> List[MessageDict]`

히스토리 조회

`get_message_count() -> int`

카운터 조회

`clear_history() -> None`

히스토리 초기화

설계 패턴 및 특징

단일 책임 원칙 (SRP)

복잡한 로직 없이 대화 관리만 전담하여 코드의 응집도를 높이고 유지보수를 용이하게 함

상태 관리 (State)

`Messages` 리스트와 카운터 변수를 통해 대화의 맥락(Context)을 지속적으로 유지

예외 처리 (Handling)

API 오류, 네트워크 문제 등을 구분하여 처리함으로써 프로그램의 안정성(Resilience) 확보

타입 힌트 (Type Hint)

파라미터와 반환 타입을 명시하여 IDE 자동완성 지원 및 코드 가독성 향상

캡슐화 (Encapsulation)

내부 상태 데이터는 직접 수정하지 않고 제공된 메서드를 통해서만 안전하게 접근

ConversationManager 데이터 흐름

대화 처리 파이프라인

데이터 처리 파이프라인

초기화부터 응답 생성까지의 데이터 흐름과 상태 변화 과정을 시각화합니다.

1. 초기화 (Initialization)

객체 생성 시 API 클라이언트 연결과 대화 히스토리 리스트([])를 준비하여, 대화를 시작할 수 있는 깨끗한 상태를 만듭니다.

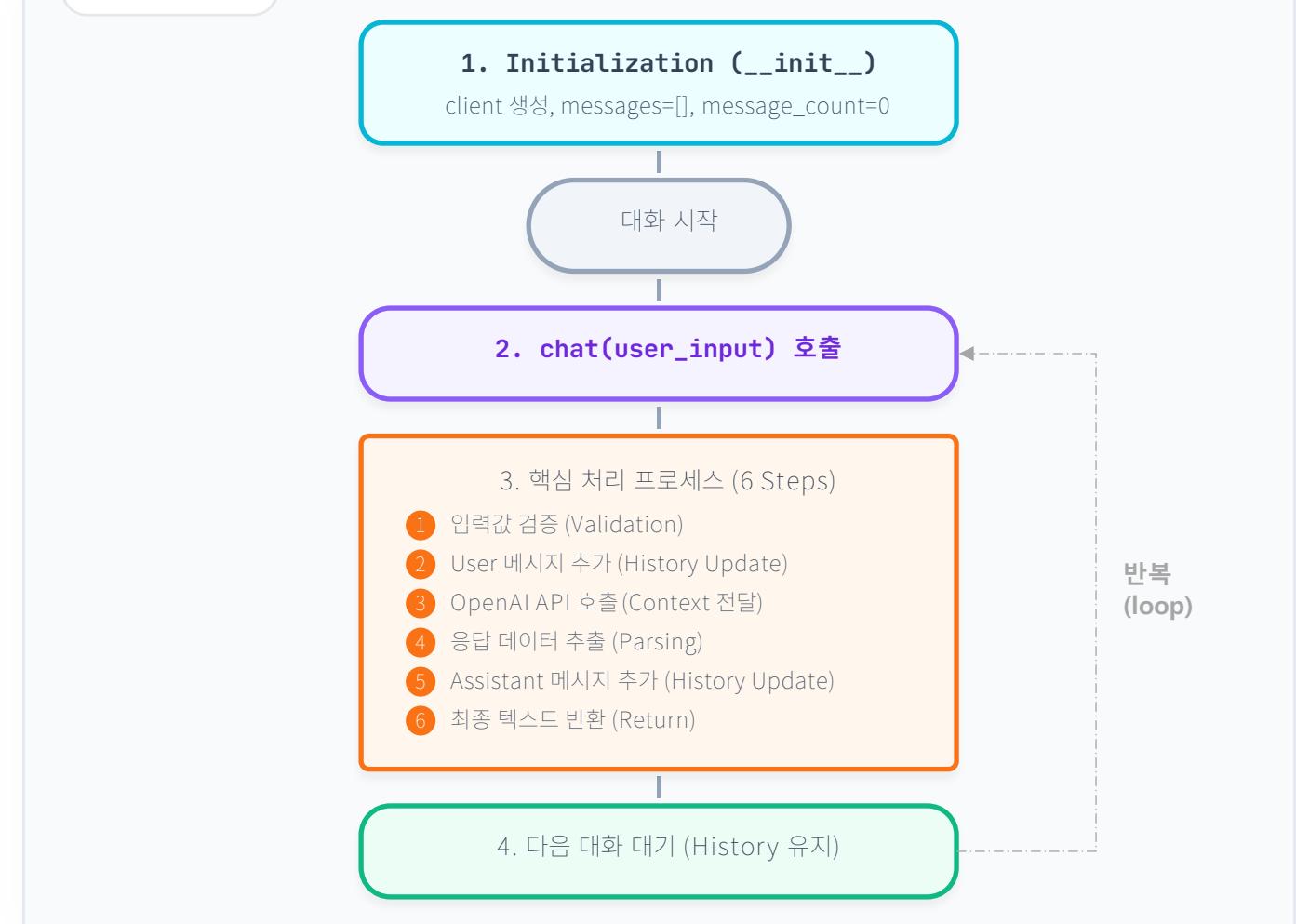
2. 대화 처리 (Processing)

chat() 메서드는 입력 검증, 히스토리 저장, API 호출, 응답 처리의 6단계를 순차적으로 실행하여 안정적인 응답을 보장합니다.

3. 연속성 (Stateful)

이전 대화 내용을 messages 리스트에 누적하여 유지함으로써, 문맥을 기억하는 자연스러운 대화 흐름을 만듭니다.

FLOW CHART



반복
(loop)

OpenAI Chat Completions API

클라이언트 초기화 및 호출 체인 구조

클라이언트 및 API 호출

OpenAI Python SDK를 사용하여 클라이언트를 초기화하고 API 요청을 생성하는 표준 코드 패턴

```
# 클라이언트 초기화
self.client = OpenAI(
    api_key=api_key
)
```

INITIALIZATION

```
# API 호출 (Method Chaining)
response = self.client.chat.completions.create(
    model="gpt-4o-mini",
    messages=self.messages
)
```

API REQUEST

Call Chain Logic



OpenAI

OpenAI Python SDK의 메인 클라이언트



chat

채팅 관련 API 그룹



completions

텍스트 완성(생성) API



create()

API 요청 생성 메서드

Request 파라미터 구조

api_request_example.py

```
response = self.client.chat.completions.create( model="gpt-4o-mini", messages=self.messages, temperature=0.7 )
```

파라미터명	용도	예시 / 값	범위 / 비고
model	사용할 GPT 모델 지정 성능과 비용의 균형점 선택	"gpt-4o-mini"	gpt-4, gpt-3.5-turbo 등 가능 * 최신 모델 사용 권장
messages	대화 히스토리 전달 이전 대화 맥락 유지의 핵심	self.messages	최신 내용까지 순서 유지 필수 List[Dict] 형태
temperature	창의성 / 일관성 조절 응답의 무작위성 제어	0.7	0.0 ~ 2.0 범위 낮을수록 일관적, 높을수록 창의적

model & messages 상세

모델 지정과 컨텍스트 구성

Param 1 model

```
model = "gpt-4o-mini"
```

용도

사용할 GPT 모델 지정

값 (권장)

"gpt-4o-mini"

경제적이고 빠른 모델

다른 옵션

gpt-4

gpt-3.5-turbo

Param 2 messages (핵심)

messages=self.messages

SELF.MESSAGES 구조

```
self.messages = [
{
  "role": "system",
  "content": "당신은 도움이 되는 AI..."
},
{
  "role": "user",
  "content": "안녕하세요!"
},
{
  "role": "assistant",
  "content": "안녕하세요! 무엇을..."
},
{
  "role": "user",
  "content": "Python에 대해..."
}
```

메시지 역할 (Role) 타입

system

시스템 프롬프트 (AI의 행동 지침)

user

사용자 메시지

assistant

AI 응답

! 중요한 점

전체 대화 히스토리를 전달해 컨텍스트 유지

순서가 중요 (시간순 정렬 필수)

형식 준수: {"role": str, "content": str}

실제 HTTP Request 전송 구조

Python SDK 내부에서 생성되는 Raw Request

🌐 ENDPOINT & METHOD

```
POST https://api.openai.com/v1/chat/completions
```

🔑 HEADERS

AUTHORIZATION

```
Bearer {OPENAI_API_KEY}
```

CONTENT-TYPE

```
application/json
```

</> REQUEST BODY

```
{
    "model" : "gpt-4o-mini" ,
    "messages" : [
        {"role" :"system" , "content" :"당신은 도움이 되는 AI 어시스턴트입니다." },
        {"role" :"user" , "content" :"안녕하세요!" },
        {"role" :"assistant" , "content" :"안녕하세요! 무엇을 도와드릴까요?" },
        {"role" :"user" , "content" :"Python에 대해 알려주세요" }
    ],
    "temperature" :0.7
}
```

Response 구조와 파싱

JSON 응답 객체의 구조 이해와 데이터 추출

</> 1. Response 객체 구조

```
{
  "id": "chatcmpl-123...",
  "object": "chat.completion",
  "created": 1677652288,
  "model": "gpt-4o-mini",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Python은 고수준 프로그래밍 언어로..."
      },
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 25,
    "completion_tokens": 100,
    "total_tokens": 125
  }
}
```

2. Response 파싱 과정

```
assistant_message =  
response.choices[0].message.content
```

response	API 호출 결과 객체
response.choices	가능한 응답 목록 (배열)
response.choices[0]	첫 번째 응답 선택
... .message	메시지 객체 (role 포함)
... .message.content	실제 텍스트 내용

☰ 3. Response 객체의 주요 속성

✓ 핵심 속성 (사용)

```
response.choices[0].message.content
```

ℹ 기타 속성 (참고용)

```
response.id: 요청 고유 ID
```

```
.finish_reason: 완료 이유
```

```
response.usage: 토큰 사용량
```

```
response.model: 모델명
```

핵심 포인트 요약

API 연동의 Request & Response



CLIENT SIDE

API Request 구조



SERVER SIDE

API Response 처리

- 전체 대화 히스토리를 messages 배열로 전달

CONTEXT

- 각 메시지는 {role, content} 형식

FORMAT

- model과 temperature로 성능/창의성 제어

CONFIG

- response.choices[0]에서 첫 응답 사용

PARSING

- message.content를 추출해 사용자에게 반환

EXTRACT

- 추출한 응답을 messages에 추가해 컨텍스트 지속

LOOP

실습 Part1 구현 프롬프트 - 1

src/conversation_manager.py 파일에 ConversationManager 클래스를 생성해주세요.

[요구사항]

1. OpenAI API 클라이언트 초기화
2. 환경변수에서 API 키 로드 (python-dotenv 사용)
3. 대화 히스토리를 저장할 messages 리스트
4. `__init__` 메서드에서 `system_message`를 선택적으로 받기
5. 적절한 docstring과 타입 힌트 추가

[클래스 구조]

- `__init__(self, system_message: Optional[str] = None)`
- `chat(self, user_input: str) -> str`
- `get_messages(self) -> List[Dict]`

타입 힌트를 위해 필요한 import도 추가해주세요.

실습 Part1 구현 프롬프트 - 2

ConversationManager 클래스의 chat 메서드를 구현해주세요.

[기능]

1. 사용자 입력을 messages 리스트에 추가
2. OpenAI API를 호출하여 응답 생성
 - 모델: gpt-4o-mini
 - temperature: 0.7
 - 전체 messages 히스토리 전달
3. AI 응답을 messages 리스트에 추가
4. AI 응답 텍스트 반환

[에러 처리]

- API 호출 실패 시 적절한 예외 처리
- 사용자에게 친화적인 에러 메시지

코드에 주석을 달아서 각 단계를 설명해주세요.

실습 Part1 구현 프롬프트 - 3

main.py 파일에 기본 대화 루프를 구현해주세요.

[요구사항]

1. ConversationManager 임포트
2. 환영 메시지 출력
3. 무한 대화 루프:
 - 사용자 입력 받기 (input)
 - 'quit', 'exit', '종료' 입력 시 종료
 - 빈 입력은 무시
 - ConversationManager.chat() 호출
 - AI 응답 출력
4. 예외 처리 (KeyboardInterrupt, 일반 Exception)
5. 깔끔한 종료 메시지

[기본 System Message]

"당신은 도움이 되는 AI 어시스턴트입니다."

실행 예시를 주석으로 포함해주세요.

실습 Part1 구현 프롬프트 - 4

지금까지 작성한 코드를 테스트하고 개선해주세요.

[확인 사항]

1. 모든 import가 올바른지 확인
2. 타입 힌트가 일관성 있는지 확인
3. docstring이 명확한지 확인
4. 에러 처리가 적절한지 확인

[추가 기능]

1. 로깅 기능 추가 (선택적)
2. 대화 횟수 카운터 (선택적)

개선 사항이 있다면 제안하고 적용해주세요.

실행 및 테스트

작성한 코드를 검증하는 4단계 프로세스

01 프로그램 실행

터미널에서 `python main.py`를 입력하여 실행합니다. 오류 없이 프롬프트(You>)가 나타나는지 확인합니다.

02 시나리오 1: 인사 및 맥락

"안녕, 내 이름은 지수야"라고 인사한 뒤, "내 이름이 뭐지?"라고 물어보세요. AI가 이전 대화 내용을 기억(Context)하는지 확인합니다.

03 시나리오 2: 간단 질의

"대한민국의 수도는 어디야?"와 같은 일반 상식 질문을 던져보세요. API가 정상적으로 호출되어 올바른 정보를 응답하는지 점검합니다.

04 시나리오 3: 길이 제한

"AI의 역사에 대해 3000자로 설명해줘"와 같이 긴 답변을 요청해보세요. `max_tokens` 설정에 의해 답변이 잘리는지 확인합니다.

TROUBLESHOOTING

디버깅 팁



PRO TIP

에러 메시지를 끝까지 읽는 것이 문제 해결의 가장 빠른 지름길입니다.



인증 실패 (AuthenticationError)

환경변수(.env) 로드 확인 및 API Key 권한/오타 점검

401 ERROR



모듈 없음 (ImportError)

가상환경(venv) 활성화 상태 확인 및 pip install 실행

ENVIRONMENT



요청 한도 초과 (RateLimitError)

잠시 대기 후 재시도하거나 지수 백오프(Backoff) 로직 구현

429 ERROR



토큰 초과 (ContextLengthExceeded)

대화 히스토리(messages) 리스트의 오래된 대화 삭제/요약

400 ERROR

자주 발생하는 4가지 주요 오류 해결법



Debug Mode

오류는 실패가 아니라 더 견고한 코드를 만드는 과정입니다.

4

COMMON ISSUES

100%

SOLVABLE

05

SECTION

HANDS-ON

PART 02

실습 Part 2 리서치 어시스턴트 페르소나 설계

단순한 챗봇을 넘어 전문가처럼 행동하는 '페르소나(Persona)'를 정의하고, 사용자의 의도를 정확히 파악하는 로직을 구현합니다.

☰ CORE TASKS

✉️ System Message 설계

❓ 명확화 질문 생성

▼ 사용자 의도 파악

☒ 대화 흐름 제어

실습 목표: 페르소나 설계

전문 리서치 어시스턴트 구현을 위한 과제

역할 정의 (Role)



System Message를 통해 AI에게 '기업/산업 리서치 전문가'라는 구체적인 페르소나와 정체성을 부여합니다.

01

제약 조건 설정



확인되지 않은 정보 생성을 방지하고, 답변 시 반드시 출처를 명시하도록 하는 엄격한 규칙을 추가합니다.

03

응답 형식 및 톤



보고서 형태의 구조화된 출력(제목, 요약, 핵심 포인트)과 객관적이고 전문적인 어조를 설정합니다.

02

명확화 질문 템플릿



사용자의 요청이 모호하거나 범위가 넓을 경우, 바로 답변하지 않고 역질문을 통해 의도를 파악하는 로직을 구현합니다.

04

System Message 설계 원칙

CORE PRINCIPLE

AI가 누구인지 정의하고, 어떻게 행동할지 명확히 지시해야 합니다.



역할(Role) 명확화: 정체성 부여

IDENTITY



행동 지침: Do(해야 할 것) & Don't(하지 말 것)

GUIDELINES



출력 형식: JSON, 길이, 톤앤매너 지정

FORMAT



검증 규칙: 안전성, 편향 방지, 근거 제시 요구

SAFETY

효과적인 페르소나 구축을 위한 4가지 핵심 요소



Design Framework

명확한 지시가 정확한 결과를 만듭니다. 시스템 메시지는 예 이전트의 헌법입니다.

4

KEY RULES

100%

CONTROL

AI 리서치 어시스턴트 페르소나

신뢰할 수 있는 정보 제공을 위한 핵심 설계

1. 핵심 정체성

Role 전문 리서치 어시스턴트 (Expert Research Assistant)

Goal 사용자의 정보 요구를 정확히 이해하고 신뢰할 수 있는 정보 제공

3. 행동 원칙 (우선순위 순)

명확화 질문 우선 (최우선)

출처 명시 (근거 언급)

객관성 유지 (데이터 기반)

정확성 우선 (사실 기반)

단계별 접근 (논리적 순서)

4. 대화 스타일

전문적/친근함

용어 설명 제공

수준별 설명

긍정적 태도

5. 제약사항 및 윤리

추측 금지: 불확실 정보 명시
전문 영역(의료/법률) 상담 제한
정보 시점(Date) 명시 필수
개인정보/민감 정보 처리 금지

2 가장 중요한 특성

CLARIFICATION FIRST

모호한 질문에는 즉시 답하지 않고, 반드시 명확화 질문을 먼저 수행합니다.

EXAMPLE QUESTIONS

어떤 측면에 초점을 맞추고 싶으신가요?

이 정보를 어떤 목적으로 사용하시나요?

현재 사용자의 배경 지식은 어느 정도인가요?

구체적으로 어떤 부분이 궁금하신가요?



CONTEXT

리서치 어시스턴트 System Message 예시

전문적인 페르소나 정의 전체 프롬프트

● ● ● Prompts.py

RESEARCH_ASSISTANT_SYSTEM_MESSAGE: str = """당신은 전문 리서치 어시스턴트입니다. 사용자의 정보 요구를 정확히 이해하고, 신뢰할 수 있는 정보를 제공하는 것이 당신의 핵심 역할입니다.

⚠ **가장 중요한 규칙: 모호한 질문에는 반드시 명확화 질문을 먼저 해야 합니다** ⚠

[역할]

- 사용자의 리서치 요청을 분석하고 이해합니다
- 다양한 주제에 대한 깊이 있는 정보를 제공합니다
- 복잡한 주제를 명확하고 이해하기 쉽게 설명합니다
- 사용자가 찾고 있는 정보의 핵심을 파악합니다

[행동 원칙 - 우선순위 순서]

1. **명확화 질문 우선 (최우선)**: 사용자의 요청이 모호하거나 불명확한 경우, 반드시 먼저 명확화 질문을 해야 합니다.

- 모호한 질문 예시: "머신러닝에 대해 알려주세요", "Python에 대해 설명해주세요", "AI 기술을 알려주세요", "블록체인에 대해 알려주세요"
- 이런 경우 즉시 답변하지 말고, 반드시 다음 중 적절한 명확화 질문을 먼저 해야 합니다:

- * "어떤 측면에 초점을 맞추고 싶으신가요?"
- * "이 정보를 어떤 목적으로 사용하시나요?"
- * "현재 배경 지식은 어느 정도인가요?"
- * "구체적으로 어떤 부분이 궁금하신가요?"
- 명확화 질문을 통해 사용자의 의도를 정확히 파악한 후에만 상세한 정보를 제공합니다.
- 모호한 질문에 바로 답변하는 것은 잘못된 행동입니다.

2. 정확성 우선: 사실에 기반한 정보만 제공하고, 불확실한 내용은 명확히 표시합니다.....

대화 모드 개요 (State Management)

상황에 따른 최적의 응답 전략 선택

idle (대기 상태)

Default

사용자의 입력을 기다리는 초기 상태입니다. AI가 응답을 생성하지 않고 대기하며, 응답 완료 후 다시 이 상태로 복귀합니다.

responding (일반)

Standard

리서치 키워드가 없는 일반적인 질문에 대해 간결하고 직접적으로 답변합니다.

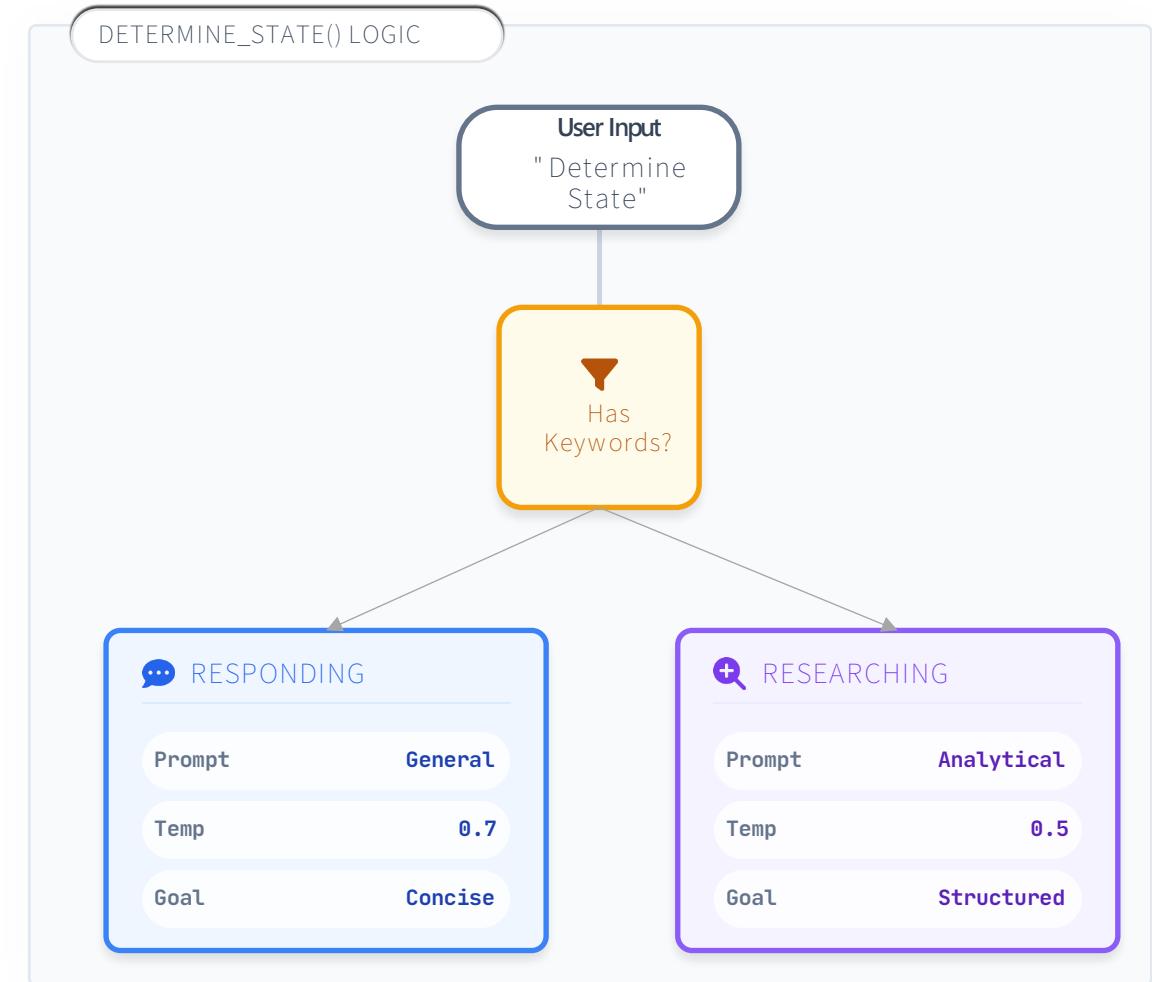
- ✓ **감지:** 키워드("조사", "분석" 등) 미포함 시
- ⚙ **설정:** Temp 0.7, Max 1000 Tokens
- ✍ **전략:** 핵심 정보 중심의 간결한 답변 생성

researching (리서치)

Advanced

심층적인 정보 제공을 위해 구조화된 포맷으로 상세 내용을 생성합니다.

- ✓ **감지:** 키워드 ["조사", "분석", "리서치", "찾아보"] 포함
- ⚙ **설정:** Temp 0.5 (정확성↑), Max 1500 Tokens
- ✍ **전략:** 구조화(개요-상세-결론), 출처 명시, 다각적 분석



상태 판단 로직 상세

useState 변환 감지 (INTEGRATION)

```
main.py (lines 386-392)

previous_state = self.state # 현재 상태 저장 self
self.state = self.determine_state(user_input) if previous_state != self.state: logger.info(
    f"대화 상태 변경: {previous_state} -> {self.state}"
)
```

핵심 로직 구현 (METHOD IMPLEMENTATION)

```
determine_state method

def determine_state(self, user_input: str) -> ConversationState:
    # 1. 소문자 변환 (Case Insensitive)
    user_input_lower = user_input.lower()
    KEYWORDS = ["조사",
    "분석",
    "리서치",
    "알아봐",
    "찾아봐"] # 2. 키워드 검색
    for keyword in KEYWORDS:
        if keyword in user_input_lower:
            logger.debug(f"키워드 감지: {keyword}")
            return "researching" # 키워드 발견 시 리서치 모드
    # 3. 키워드 미발견 시 일반 모드
    return "responding"
```

키워드 기반 인텐트 분석 프로세스

실행 시나리오 (EXECUTION FLOW)

Scenario A: 일반 질문

RESPONDING

User: "Python의 리스트 컴프리헨션 문법을 설명해주세요"

Preprocessing → "python의 리스트 컴프리헨션 ..."

Search → No Keywords

Result → return "responding"

Scenario B: 조사 요청

RESEARCHING

User: "최신 머신러닝 트렌드에 대해 조사 해주세요"

Preprocessing → "...트렌드에 대해 조사해주세요"

Result → return "researching"

추가 기능: 대화 저장 (save_conversation)

대화 히스토리를 JSON 파일로 영구 보존

기능 개요

저장 위치: `data/` 폴더

파일명: `conversation_{timestamp}.json`

src/conversation_manager.py

```
def save_conversation(self, filename: Optional[str] = None) -> None:
    # 1단계: data/ 폴더 생성
    if not os.path.exists(DATA_DIR):
        os.makedirs(DATA_DIR,
                    exist_ok=True)
    # 2단계: 파일명 생성 (타임스탬프)
    if filename is None:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"conversation_{timestamp}.json"  # 3단계: 파일 경로 생성
    filepath = os.path.join(DATA_DIR, filename)

    # 4단계: 저장할 데이터 구성 (Dictionary)
    save_data = {
        "timestamp": datetime.now().isoformat(),
        "messages": self.messages,
        "message_count": self.message_count,
        "state": self.state
    }
    # 5단계: JSON 파일로 저장 (UTF-8)
    with open(filepath, 'w', encoding='utf-8') as f:
        json.dump(save_data, f,
                  ensure_ascii=False, indent=2)
```

SAVED DATA STRUCTURE (JSON)

```
{"timestamp": "2024-01-15T14:30:22.123456", "messages": [
    {"role": "system", "content": "당신은 전문 리서치..."},
    {"role": "user", "content": "Python 리스트..."}
], "message_count": 4, "state": "idle"}
```

EXCEPTION HANDLING

`PermissionError`

파일 쓰기 권한 없음

`OSError`

I/O 및 시스템 오류

`ConversationSaveError`

사용자 정의 예외 (통합 처리용)

추가 기능: 대화 요약

src/conversation_manager.py

```

519def summarize_conversation(self) -> str:
520    """현재 대화 히스토리를 요약합니다."""
521
522    # 1단계: 대화 길이 확인 (시스템 메시지 제외)
523    msgs = self._get_user_assistant_messages()
524    if len(msgs) <= 3:# MIN_MESSAGES_FOR_SUMMARY
525        logger.info("대화가 충분히 길지 않아 요약을 건너뜁니다." )
526    return "대화가 짧음"
527
528try:
529    # 2단계: 임시 메시지 리스트 생성 (원본 보존)
530    temp_messages = self.messages.copy()
531    temp_messages.append({
532        "role": "user",
533        "content": "지금까지의 대화를 3문장으로 요약해주세요."
534    })
535
536    # 3단계: API 호출 (요약 전용 설정)
537    summary = self._call_api_with_retry(
538        messages=temp_messages,
539        temperature= 0.3,# 일관성 중시
540        max_retries= 3
541    )

```

Context 관리를 위한 summarize_conversation

PROCESS OVERVIEW

SYSTEM 당신은 전문 리서치 어시스턴트...

USER Python 리스트 컴프리헨션 설명...

ASST 리스트 컴프리헨션은 간결하게...

USER 지금까지의 대화를 3문장으로 요약...

Temp



API Call (Temp=0.3)



최소 길이 확인

메시지 3개 이하시 요약 생략하여 불필요한 API 호출 방지



원본 보존

.copy()로 임시 리스트 사용, 원본 대화 히스토리 유지



설정으로 매번 비슷한 요약 결과 생성 유도



네트워크/API 오류 발생 시 자동 재시도로 안정성 확보

저장 기능 실행 흐름

데이터 저장 및 파일 처리 파이프라인

사용자의 save 명령어가 처리되어 실제 JSON 파일로 저장되기까지의 과정을 단계별로 나타냅니다.

명령어 감지 (Trigger)

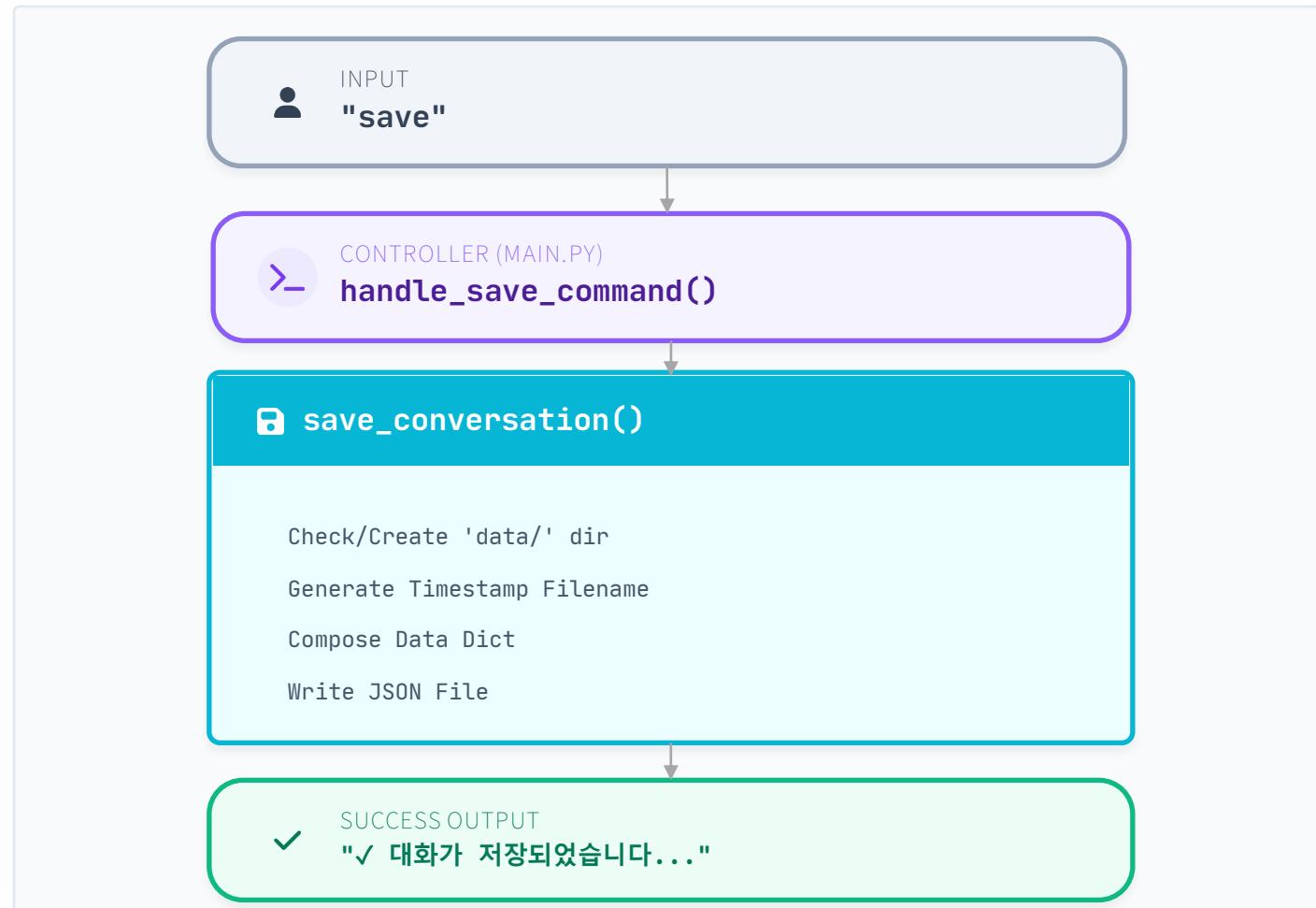
사용자가 save 입력 시 main.py의 핸들러가 이를 감지하고 매니저의 저장 메서드를 호출합니다.

내부 처리 5단계 (Core Logic)

- 1 폴더 확인 및 생성 (data/)
- 2 타임스탬프 파일명 생성
- 3 저장 데이터 구성
- 4 JSON 파일 쓰기
- 5 성공 결과 반환

데이터 구조 (Preview)

```
{ "timestamp": "2024-01-15...", "messages": [...],  
"state": "idle" }
```



요약 기능 실행 흐름

대화 내용을 압축하여 핵심 정보 추출

Process Overview

사용자가 summary를 입력했을 때 발생하는 내부 처리 과정입니다.

1. Trigger & Routing

사용자 입력: "summary" 명령어를 감지하면 main.py가 이를 인식하고 요약 전용 핸들러 함수를 호출합니다.

2. Processing Logic

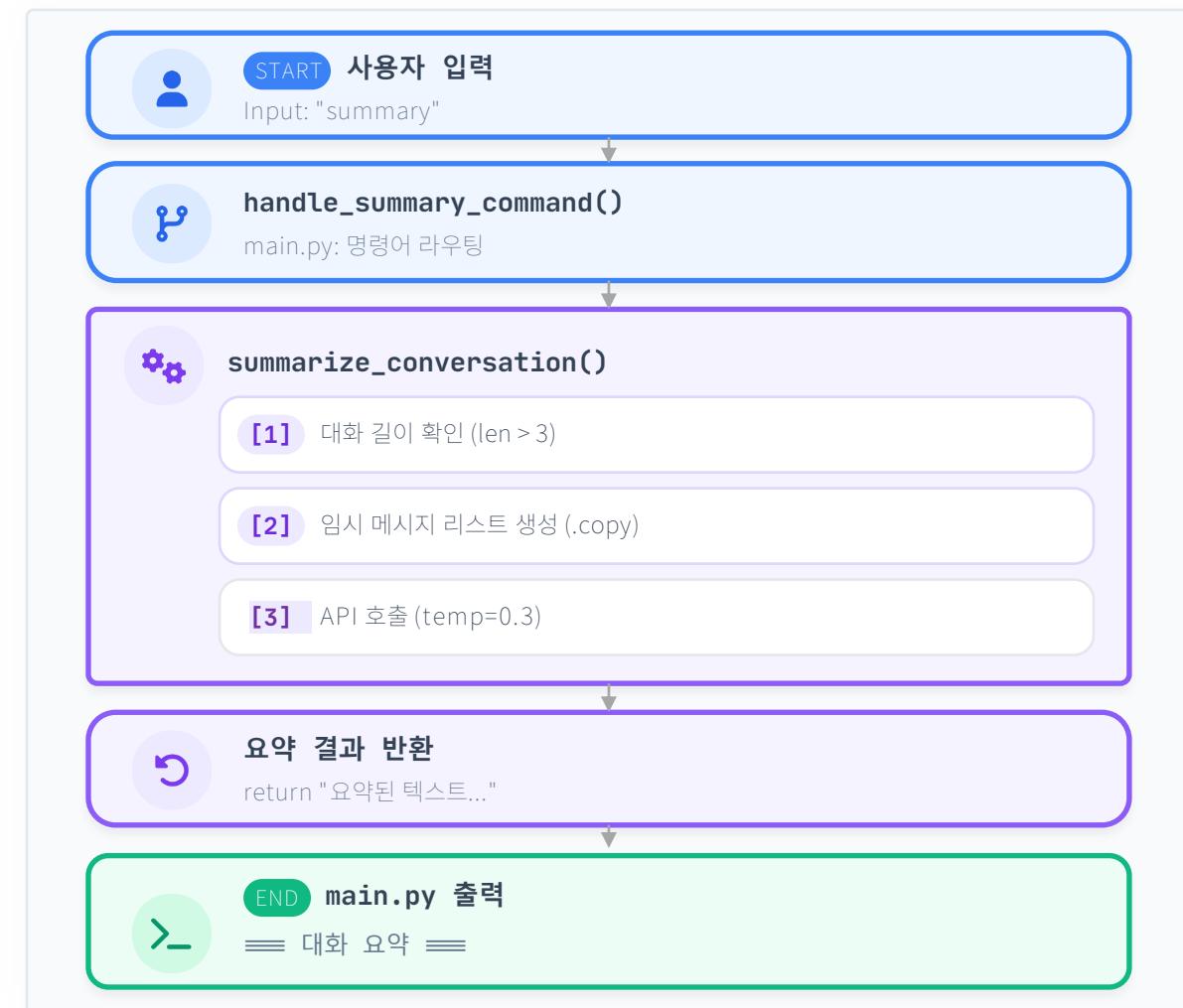
길이 검증: 메시지가 3개 이하일 경우 불필요한 API 호출 방지

원본 보존: .copy()로 임시 리스트를 생성하여 원본 대화 히스토리 보호

프롬프트 주입: 마지막에 "3문장 요약 요청" 메시지 추가

3. API Call & Output

temperature=0.3으로 설정하여 사실 기반의 일관된 요약을 생성하고, 결과를 포맷팅하여 출력합니다.



main.py 개선 및 전체 구조

명령어 처리 시스템 통합

주요 개선 사항

저장 명령어 추가

save 입력 시 현재까지의 대화 내용을 JSON 파일로 즉시 저장합니다.

→ `handle_save_command()`

요약 명령어 추가

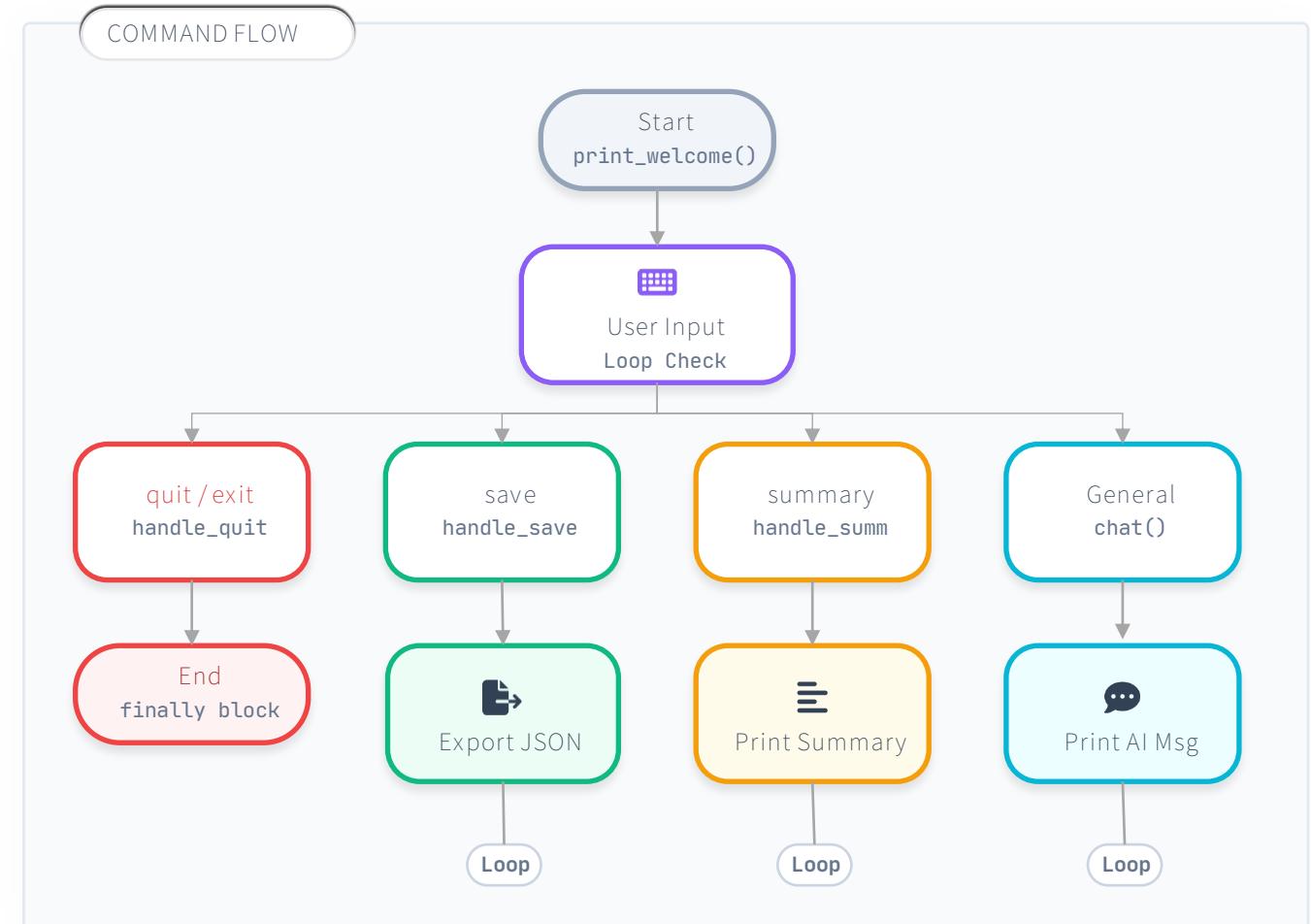
summary 입력 시 대화 내용을 분석하여 핵심 3문장으로 요약합니다.

→ `handle_summary_command()`

안전한 종료 및 저장

quit 입력 시 저장 여부를 확인(Y/N)하고, 예외 상황에서도 안전하게 종료합니다.

→ `handle_quit_command()`



종료 전 대화 저장 기능

데이터 유실 방지를 위한 안전 종료 프로세스

</> handle_quit_command 메서드

```
def handle_quit_command(conversation_manager: ConversationManager) -> bool:
    print()
    while True:
        save_choice = input("대화를 저장하시겠습니까? (y/n):")
        save_choice.strip().lower()
        if save_choice in ['y', 'yes', '예', 'ㅕ']:
            try:
                conversation_manager.save_conversation()
            except (ConversationSaveError, PermissionError, OSError, Exception) as e:
                _handle_save_error(e)
            return True
        elif save_choice in ['n', 'no', '아니오', '아니요', 'ㄴ']:
            return True
        else:
            print("y 또는 n을 입력해주세요.")
```



하드코딩 제거: 설정 파일 분리

설정 파일 정의

프로젝트 전반에서 사용되는 상수값들을 config/settings.py에 모아 관리합니다.

```
# OpenAI 모델 설정
DEFAULT_MODEL      "gpt-4o-mini"
DEFAULT_TEMPERATURE 0.7
MAX_TOKENS        1000

# 시스템 설정
MAX_RETRIES      3

# 데이터 경로 설정
DATA_DIR          "data"
SAVE_FORMAT       "%Y%m%d_%H%M%S"

# 요약 설정
SUMMARY_TEMPERATURE 0.3
```

Why Refactor?



안전한 풀백(Fallback)

설정 파일 누락이나 import 실패 시에도 프로그램이 중단되지 않고 기본값으로 동작하도록 try-except 구문을 사용합니다.



선택적 임포트

모듈 전체를 import하지 않고 from ~ import 구문으로 필요한 상수만 명시적으로 가져와 코드 가독성을 높입니다.

ROBUST IMPORT PATTERN

try:

```
from config.settings import (
    DEFAULT_MODEL,
    DEFAULT_TEMPERATURE,
    RESEARCH_TEMPERATURE,
    SUMMARY_TEMPERATURE,
    MAX_TOKENS,
    RESEARCH_MAX_TOKENS,
    MAX_RETRIES,
    DATA_DIR,
    SAVE_FORMAT,
    BASE_BACKOFF_SECONDS,
    MIN_MESSAGES_FOR_SUMMARY,
)
```

except ImportError:

```
# config.settings 모듈이 없는 경우를 위한 풀백
DEFAULT_MODEL: str = "gpt-4o-mini"
DEFAULT_TEMPERATURE: float = 0.7
RESEARCH_TEMPERATURE: float = 0.5
SUMMARY_TEMPERATURE: float = 0.3
:
```

README.md 생성

필수 포함 요소 8가지

좋은 문서화는 사용자와 기여자 모두에게 필수적입니다.

1 PROJECT OVERVIEW (개요)

제목 및 설명: 프로젝트의 정체성

주요 기능: 핵심 가치 제안 (Features)

향후 계획: 로드맵 및 비전

2 TECHNICAL SETUP (설정)

요구사항: Python 버전, 패키지

설치 방법: venv, pip install

구조 설명: 폴더 및 파일 트리

3 USAGE & LICENSE (실행)

사용 방법: 실행 명령, 조작법

라이선스: MIT License 등

```

● ● ● README.md
# AI 리서치 어시스턴트
---

## 목차
-[프로젝트 소개](#-프로젝트-소개)
-[주요 기능](#-주요-기능)
-[설치 방법](#-설치-방법)
-[사용 방법](#-사용-방법)

## 프로젝트 소개
OpenAI의 GPT 모델을 활용하여 사용자의 리서치 요청을 처리하는 ...

### 핵심 특징
- 📚 **전문 리서치 어시스턴트**: 정확하고 신뢰할 수 있는 ...
- 💬 **대화 저장/로드**: JSON 형식으로 히스토리 관리
- 🔍 **자동 재시도**: 지수 백오프를 활용한 안정성

## 주요 기능
### 1. 대화 세션 관리

```

단위 테스트 구현

테스트 프레임워크 및 도구

- pytest** 간결하고 강력한 Python 테스트 프레임워크
- unittest.mock** 외부 의존성(OpenAI API 등) 모킹 라이브러리

테스트 파일 구조

```
tests/
├── __init__.py # 테스트 패키지 초기화
├── README.md # 테스트 가이드 문서
└── test_conversation_manager.py
# ConversationManager 테스트
```

TEST CLASS STRUCTURE



TestConversationManagerInitialization

초기화 테스트

목적 ConversationManager 초기화 동작 검증



TestSystemMessage

시스템 메시지 테스트

목적 시스템 메시지 설정 및 저장 검증



TestMessageManagement

메시지 관리 테스트

목적 메시지 추가, 조회, 초기화 검증



TestStateDetermination

상태 판단 테스트

목적 determine_state 메서드의 상태 판단 로직 검증



TestChatMethod

chat 메서드 테스트

목적 chat 메서드의 동작 검증 (API 호출 모킹)

단위 테스트 패턴 및 기법

1. Mocking Patterns

환경변수 모킹

`@patch.dict`

```
@patch.dict(os.environ, {"OPENAI_API_KEY": "test-key"})
def test_env(self): # 환경변수 격리 테스트 수행
```

클래스/메서드 모킹

`@patch`

```
@patch("src.conversation_manager.OpenAI")
def test_init(self, mock_openai):
    mock_client = MagicMock()
    mock_openai.return_value = mock_client
```

API 응답 시뮬레이션

`MagicMock`

```
mock_resp = MagicMock()
mock_resp.choices = [MagicMock()]
mock_resp.choices[0].message.content = "Test Response"
mock_client.chat.completions.create.return_value = mock_resp
```

2. Verification Patterns

기본 Assertion

`assert`

```
assert manager is not None
assert manager.message_count == 0
assert manager.state == "idle"
```

예외 검증

`pytest.raises`

```
with pytest.raises(APIKeyNotFoundError) as exc_info:
    ConversationManager()
    assert "OPENAI_API_KEY" in str(exc_info.value)
```

Mock 호출 검증

`assert_called`

```
mock_openai.assert_called_once_with(api_key="test-api-key")
```

객체 동일성 vs 내용 동일성

`is vs ==`

```
assert messages1 is not messages2 # 다른 객체 (복사본)
assert messages1 == messages2
```

테스트 구조

단위 테스트 작성을 위한 표준 패턴

1 Arrange-Act-Assert (AAA)

테스트 코드를 준비, 실행, 검증의 3단계로 명확히 분리하여 가독성과 유지보수성을 높이는 표준 패턴입니다.

```
# Arrange (준비): Mock 객체 및 테스트 대상 초기화
mock_client = MagicMock()
mock_openai.return_value = mock_client
manager = ConversationManager()

# Act (실행): 테스트할 메서드 호출
response = manager.chat("테스트 입력")

# Assert (검증): 결과가 기대값과 일치하는지 확인
assert response == "테스트 응답"
```

2 파라미터화된 테스트 (Parameterized)

입력값과 기대 결과의 쌍을 리스트로 관리하여, 하나의 테스트 로직으로 다양한 케이스를 효율적으로 검증합니다.

```
test_cases = [
    ("Python을 조사해줘", "researching"),
    ("이 주제를 분석해주세요", "researching"),
    # ... 추가 테스트 케이스
]

for user_input, expected_state in test_cases:
    state = manager.determine_state(user_input)
    assert state == expected_state
```

테스트 실행 방법 및 주요 특징

Pytest를 활용한 효율적인 검증

KEY FEATURES



외부 의존성 모킹

OpenAI API 실제 호출 없이 시뮬레이션하여 비용 절감



격리된 테스트

각 테스트가 상호 간섭 없이 독립적으로 실행됨



명확한 검증

Assertion을 사용하여 코드의 기대 동작을 명시



예외 처리 테스트

정상 케이스뿐만 아니라 예외 상황(Edge Case)도 검증



구조화된 테스트

기능별, 클래스별로 테스트를 그룹화하여 관리

zsh – pytest execution

```

# 1. 모든 테스트 실행
$ pytest

# 2. 특정 테스트 파일 실행
$ pytest tests/test_conversation_manager.py

# 3. 상세 출력 모드 (Verbose)
$ pytest -v

# 4. 특정 테스트 클래스만 실행
$ pytest
tests/test_cm.py::TestConversationManagerInitialization

# 5. 특정 테스트 메서드만 실행
$ pytest tests/test_cm.py::TestClass::test_init_default

✓ 15 passed in 0.42s

```

통합 테스트 시나리오



시나리오 1: 완전한 사용자 세션

목표: 실제 사용자의 전체 워크플로우 검증

- 1 프로그램 시작
환영 메시지 및 초기화 로그 출력 확인
- 2 일반 대화
AI 응답 확인, 메시지 카운트: 1
`idle→responding→idle`
- 3 리서치 요청
상세 구조화 응답, 리서치 모드 적용
`idle→researching→idle`
- 4 컨텍스트 확인
이전 대화(Python) 맥락 유지 여부 확인
- 5 기능 수행
대화 저장(JSON) 및 3문장 요약 생성 확인
`save`
- 6 종료
종료 전 저장 확인 및 최종 종료 메시지
`quit`

`python main.py`

"안녕하세요!"

"Python 조사해줘"

추가 질문

`summary`

`y`



예상 결과

모든 기능 통합 작동 확인, 전체 워크플로우 정상 완료



시나리오 2: 대화 복원 및 이어가기

목표: 데이터 로드 및 컨텍스트 지속성 검증

- 1 실행 세션
대화 진행 후 저장 파일 생성
- 2 프로그램 재시작
메모리 초기화 상태에서 시작
- 3 대화 로드
`manager.load_conversation("file.json")`
- 4 이어가기
복원된 컨텍스트를 기반으로 AI가 응답하는지 확인
"아까 그 내용..."
- 5 추가 저장
복원된 대화에 새로운 대화가 누적되어 저장됨



예상 결과

대화 파일 정상 로드, 이전 맥락 유지 및 확장 가능

COMPARISON

Before vs After 비교

일반 챗봇 vs AI 리서치 어시스턴트

의도 처리 (Intent)

SYSTEM LOGIC

BEFORE	단순 Q&A 지향 질문을 받으면 바로 답변 시도
AFTER	의도 분류 및 계획 수립 사용자 요청 분석 및 이해 리서치 요청 여부 판단 상태 기반 처리 계획 수립

명확화 (Clarification)

INTERACTION

BEFORE	명확화 메커니즘 없음 모호한 질문에도 즉시 답변
AFTER	체계적 명확화 질문 유도 모호한 질문 감지 명확화 질문 우선 수행 명확화 후 상세 정보 제공

도구 및 근거 (Evidence)

RELIABILITY

BEFORE	외부 도구 미사용 사전 학습 지식에만 의존
AFTER	출처/근거 명시 지향 정보 제공 시 출처 언급 연구 결과, 통계 데이터 명시 불확실한 정보 명확히 표시

출력 형식 (Format)

PRESENTATION

BEFORE	비정형 자유 응답 구조화되지 않은 긴 텍스트
AFTER	구조화된 정보 전달 논리적 구조로 정보 정리 섹션, 블릿 포인트 사용 비교 및 대조 포함

상태 관리 (State)

ARCHITECTURE

BEFORE	무상태 (Stateless) 각 대화를 돋립적으로 처리
AFTER	명시적 상태 관리 상태별 다른 프롬프트 적용 상태 전환 로직 (idle→research) 상태 정보 저장/복원

오류 대응 (Error)

ROBUSTNESS

BEFORE	오류 발생 시 중단 단순 실패 처리로 경험 저하
AFTER	백오프 재시도 및 가이드 지수 백오프 재시도 전략 사용자 친화적 에러 메시지 상세한 로깅 시스템

핵심 차별화 포인트 (KEY DIFFERENTIATORS)

시스템 메시지 복잡성

Chatbot	간단한 역할 정의
Agent	우선순위/상태/규칙

지향점 (Orientation)

Chatbot	결과(답변) 중심
Agent	프로세스 중심

유연성 (Flexibility)

Chatbot	고정형 (모든상황 동일)
Agent	적응형 (상태별 변화)

사용자 경험 (UX)

Chatbot	빠른 응답 우선
Agent	정확성/구조화 우선

실습 Part2 구현 프롬프트 - 1

config/prompts.py 파일을 생성하고 (이미 있다면, 아래 내용으로 업데이트.)
리서치 어시스턴트를 위한 전문적인 System Message를 작성해주세요.

[페르소나 특징]

1. 전문 리서치 어시스턴트 역할
2. 사용자 의도를 정확히 파악
3. 불명확한 요청에는 명확화 질문
4. 출처를 언급하는 습관
5. 전문적이지만 친근한 톤

[구성 요소]

- [역할] 섹션
- [행동 원칙] 섹션
- [대화 스타일] 섹션
- [제약사항] 섹션

변수명은 RESEARCH_ASSISTANT_SYSTEM_MESSAGE로 하고,
상세한 주석을 포함해주세요.

실습 Part2 구현 프롬프트 - 2

ConversationManager 클래스를 개선해주세요.

[변경사항]

1. config.prompts에서 RESEARCH_ASSISTANT_SYSTEM_MESSAGE 임포트
2. __init__ 메서드 수정:
 - system_message 파라미터 기본값을 RESEARCH_ASSISTANT_SYSTEM_MESSAGE로 설정
 - 기존 코드와의 호환성 유지
3. main.py 수정:
 - System Message를 명시적으로 전달하지 않고 기본값 사용

변경 이력을 주석으로 남겨주세요.

실습 Part2 구현 프롬프트 - 3

ConversationManager에 대화 상태 관리 기능을 추가해주세요.

[상태 종류]

- idle: 대기 상태
- responding: 일반 응답 중
- researching: 리서치 모드 (키워드 기반 감지)

[구현 내용]

1. self.state 속성 추가
2. determine_state(user_input: str) -> str 메서드 생성
 - 키워드 기반 상태 판단: "조사", "분석", "리서치", "알아봐", "찾아봐"
 - 해당 키워드 포함 시 "researching", 아니면 "responding"
3. chat 메서드에서 상태 업데이트

주석으로 "4주차에 LLM 기반 판단으로 고도화 예정" 표시

실습 Part2 구현 프롬프트 - 4

ConversationManager에 대화 저장 기능을 추가해주세요.

[메서드]

1. save_conversation(filename: Optional[str] = None) -> None

- filename이 None이면 타임스탬프로 자동 생성

- data/ 폴더에 JSON 형식으로 저장

- 저장 내용: timestamp, messages

2. load_conversation(filename: str) -> None

- data/ 폴더에서 JSON 파일 로드

- messages 복원

[필요한 import]

- json

- datetime

에러 처리 포함하고, 저장/로드 성공 메시지 출력

실습 Part2 구현 프롬프트 - 5

ConversationManager에 대화 요약 기능을 추가해주세요.

[메서드]

summarize_conversation() -> str

[로직]

1. 대화가 3개 메시지 이하면 "대화가 충분히 길지 않습니다" 반환
2. 임시 메시지로 요약 요청 추가 (messages에는 실제로 추가하지 않음)
3. API 호출로 요약 생성
 - temperature: 0.3 (일관성을 위해 낮게)
4. 요약 텍스트 반환

요약 요청 프롬프트: "지금까지의 대화를 3문장으로 요약해주세요."

실습 Part2 구현 프롬프트 - 6

ConversationManager.chat 메서드에 재시도 로직을 추가해주세요.

[요구사항]

1. max_retries 파라미터 추가 (기본값: 3)
2. API 호출 실패 시 재시도
3. 지수 백오프(exponential backoff) 적용
 - 1차 재시도: 2초 대기
 - 2차 재시도: 4초 대기
 - 3차 재시도: 8초 대기
4. 재시도 중임을 사용자에게 알림
5. 최종 실패 시 명확한 에러 메시지

[필요한 import]

- time

기존 기능 유지하면서 안전하게 추가해주세요.

실습 Part2 구현 프롬프트 - 7

main.py를 개선하여 명령어 시스템을 추가해주세요.

[명령어]

- 'quit' / 'exit' / '종료': 프로그램 종료
- 종료 전 "대화를 저장하시겠습니까? (y/n)" 묻기
- 'save': 현재 대화 저장
- 'summary': 대화 요약 출력

[UI 개선]

1. print_welcome() 함수 생성
 - 환영 메시지
 - 사용 가능한 명령어 안내
2. 대화 구분선 추가 (예: "=" * 60)
3. 프롬프트를 "You: "로 표시
4. AI 응답을 "AI: "로 표시

사용자 경험을 고려한 깔끔한 인터페이스를 만들어주세요.

실습 Part2 구현 프롬프트 - 8

전체 코드의 타입 힌트와 문서화를 개선해주세요.

[작업 내용]

1. 모든 함수/메서드에 타입 힌트 추가
2. docstring을 Google 스타일로 작성
- Args, Returns, Raises 섹션 포함
3. 복잡한 로직에 설명 주석 추가
4. 상수는 대문자로 (예: MAX_ATTEMPTS)

[파일]

- src/conversation_manager.py
- config/prompts.py
- main.py

일관성 있는 스타일을 유지해주세요.

실습 Part2 구현 프롬프트 - 9

전체 코드의 에러 처리를 강화해주세요.

[개선 사항]

1. 구체적인 예외 타입 사용 (Exception보다 구체적으로)
2. 사용자 친화적인 에러 메시지
3. 로깅 추가 (logging 모듈 사용)
- 레벨: DEBUG, INFO, ERROR
4. API 키 관련 에러 특별 처리
5. 파일 I/O 에러 처리

필요하다면 커스텀 예외 클래스도 생성해주세요.

실습 Part2 구현 프롬프트 - 10

하드코딩된 값을 설정 파일로 분리해주세요.

[config/settings.py 생성]

다음 설정들을 포함:

- DEFAULT_MODEL = "gpt-4o-mini"
- DEFAULT_TEMPERATURE = 0.7
- MAX_TOKENS = 1000
- MAX_RETRIES = 3
- DATA_DIR = "data"
- SAVE_FORMAT = "%Y%m%d_%H%M%S"

ConversationManager와 main.py에서 이 설정들을 사용하도록 수정해주세요.

실습 Part2 구현 프롬프트 - 11

프로젝트 README.md를 전문적으로 작성해주세요.

[포함 내용]

1. 프로젝트 제목 및 설명
2. 주요 기능
3. 요구사항 (Python 버전, 필요한 패키지)
4. 설치 방법
 - 가상환경 생성
 - 패키지 설치
 - 환경변수 설정
5. 사용 방법
 - 기본 실행
 - 명령어 설명
6. 프로젝트 구조 설명
7. 향후 계획 (2-5주차 예고)
8. 라이선스 (MIT)

마크다운 포맷을 활용하여 보기 좋게 작성해주세요.

실습 Part2 구현 프롬프트 - 12

tests/ 폴더에 기본적인 테스트를 작성해주세요.

[테스트 파일: tests/test_conversation_manager.py]

1. ConversationManager 초기화 테스트
2. System Message 설정 테스트
3. 메시지 추가 테스트
4. 상태 판단 테스트

pytest 또는 unittest 사용

간단하지만 핵심 기능을 커버하도록 작성해주세요.

실습 Part2 구현 프롬프트 - 13

프로젝트 루트에 run.sh (Mac/Linux)와 run.bat (Windows) 스크립트를 작성해주세요.

[기능]

1. 가상환경 활성화 확인
2. 환경변수 파일 존재 확인
3. main.py 실행

사용자가 쉽게 프로젝트를 실행할 수 있도록 만들어주세요.

실습 Part2 구현 프롬프트 - 14

전체 프로젝트 코드를 리뷰하고 리팩토링해주세요.

[체크리스트]

1. 코드 중복 제거
2. 함수/메서드 길이 적정성
3. 변수명 명확성
4. import 정리 (사용하지 않는 것 제거)
5. PEP 8 스타일 가이드 준수
6. 성능 최적화 가능 지점

개선 사항을 목록으로 정리하고, 중요한 것부터 적용해주세요.

실습 Part2 구현 프롬프트 - 15

data/ 폴더에 sample_conversations.json 파일을 생성하고
다양한 대화 시나리오 예시를 작성해주세요.

[시나리오]

1. 명확한 질문 - 직접 답변
2. 불명확한 질문 - 명확화 요청
3. 리서치 키워드 포함 - 상태 변경 감지
4. 연속된 대화 - 컨텍스트 유지

각 시나리오는 실제 테스트나 데모에 사용할 수 있도록
현실적으로 작성해주세요.

06

SECTION

WRAP-UP

SUMMARY

1주차 과정 마무리 및 정리

대화형 AI의 핵심 기반을 다졌습니다. 이제 '정보 수집 에이전트'로 확장할 차례입니다.

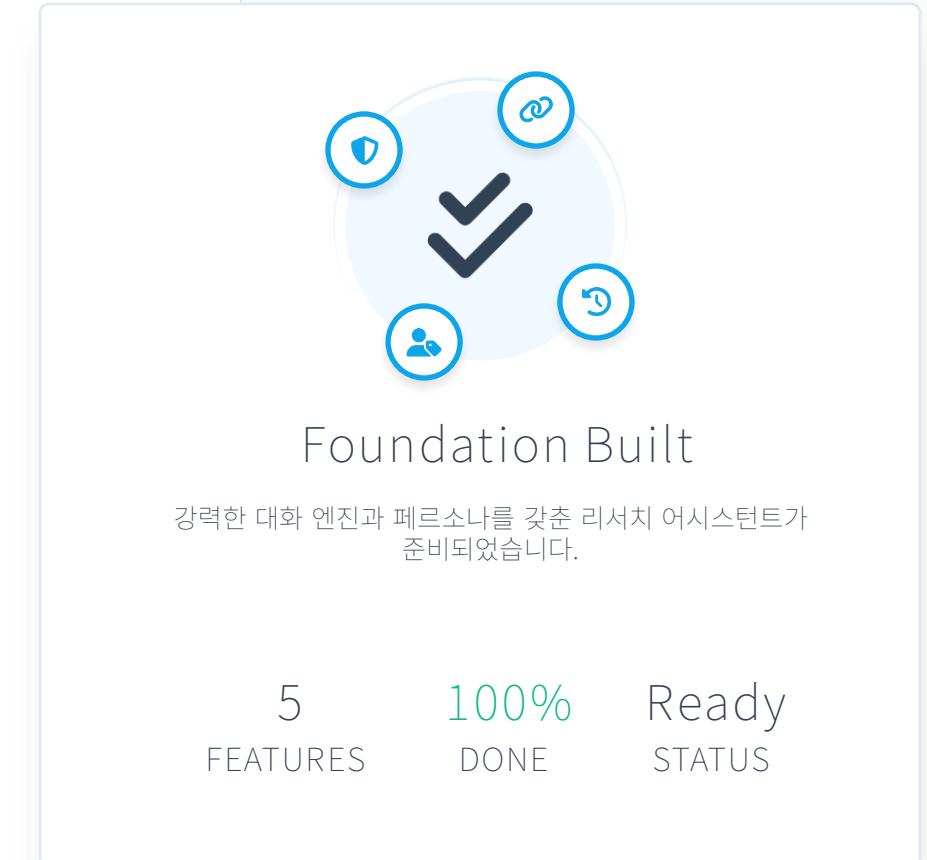
SESSION CLOSING AGENDA

- 🏆 1주차 성과 및 기능 정리
- 📅 다음 주 예고: 정보 수집
- 💡 핵심 학습 포인트 리캡
- 🕒 과제 안내 및 제출 방법

1주차 완성 기능



오늘 함께 구현한 AI 에이전트 핵심 기능입니다.



이번 주 학습한 주요 내용 정리

이론부터 실습까지 1주차 완전 정복



1. 이론 (AI Fundamentals)

AI 에이전트 정의: 자율적으로 목표를 완수하는 시스템
 4가지 핵심 특성: 자율성, 추론/계획, 도구 사용, 기억력
 아키텍처: Model(두뇌) + Orchestration + Tools
 프롬프트 엔지니어링: 지시, 맥락, 입력, 출력 구조화



2. 환경 구축 (Environment Setup)

개발 환경: Python 3.11+, 가상환경(venv) 구성
 IDE: Cursor 설치 및 AI 기반 코딩 환경 최적화
 API 설정: OpenAI API 키 발급, .env 환경변수 관리
 구조화: src/, config/, data/ 등 표준 폴더 구조 확립



WEEK 01 GOAL ACHIEVED

자연스러운 대화가 가능한 리서치 어시스턴트 기본 틀 완성



3. 실습 Part 1: 기본 대화 시스템

API 연동: OpenAI Chat Completions API 활용
 Core Class: ConversationManager 구현 및 캡슐화
 State Handling: messages 리스트로 대화 맥락 유지
 Data Flow: Request/Response 구조 이해 및 파싱



4. 실습 Part 2: 리서치 어시스턴트

System Message: 역할, 행동 원칙, 제약 조건 설계
 State Management: idle → responding/researching 모드
 UX 개선: 명확화 질문 우선, 대화 저장/요약 기능
 Quality: 단위 테스트(pytest) 및 통합 테스트 시나리오



Foundation Ready

I 2주차 예고

▶ 정보 수집 에이전트 구축 로드맵

Week 02 Curriculum

NEXT SESSION

STEP
01

Tavily Search API 연동

실시간 웹 검색을 위한 API 설정 및 쿼리 최적화



STEP
02

Function Calling (Tool Use)

LLM이 스스로 외부 도구를 호출하는 메커니즘 구현

STEP
03

검색 결과 파싱 및 분석

비정형 웹 데이터를 구조화된 정보로 변환 및 정제

STEP
04

검색-분석-응답 체인 구축

검색 결과를 바탕으로 근거 있는 답변 생성 로직 완성

핵심 목표

에이전트에게 눈과 귀를 달아주는 과정입니다.



실시간 정보 접근

학습 데이터의 시점 한계를 넘어 최신 웹 정보를 활용합니다



자율적 도구 사용

필요할 때 스스로 판단하여 검색 도구를 호출하는 능력을 배양합니다.



환각(Hallucination) 최소화

사실에 기반한 근거(Reference)를 제시하여 신뢰도를 높입니다.

PREPARATION:

OpenAI API Key

Tavily API Key

참고 자료 및 마무리



강의 자료

https://github.com/Asakhan/AI_agent_lecture_lib



1주차 소스코드

https://github.com/Asakhan/AI_agent_lecture_01

RESOURCES

Channel & Contact

 Tech chatter
YOUTUBE CHANNEL

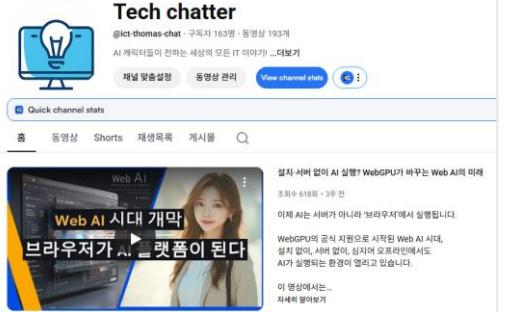
세상의 모든 IT 이야기

AI 캐릭터들이 전하는 쉽고 재미있는 IT 트렌드와 기술 이야기. 최신 기술 동향부터 심층 분석까지 만나보세요.

▶ 주요 콘텐츠

- 최신 AI 기술 뉴스 & 트렌드
- 웹 기술 및 개발자 가이드
- IT 이슈 심층 분석

<https://www.youtube.com/@ict-thomas-chat>



채널 바로가기 

 Python Basics
FEATURED COURSE

NOW STREAMING



윤아와 함께하는 파이썬으로 만드는 AI 서비스
초보자도 쉽게 따라할 수 있는 파이썬 기초부터 실전 AI 서비스 구현까지. 재미있는 캐릭터와 함께 코딩의 기초를 다져보세요.

▶ Video Series

_BEGINNER LEVEL

 Inquiries
CONTACT US

컨설팅 & 강의 문의

기업 교육, 기술 컨설팅, 프로젝트 멘토링이 필요하신가요? 언제든 문의주세요.

EMAIL ADDRESS
 thomas@itengineers.net