# RDF 온톨로지와 벡터 검색을 활용한
# 동적 멀티 에이전트 워크플로우 시스템

RDF Knowledge Graph + FAISS Vector Search + LangChain Agent

2026.01.02

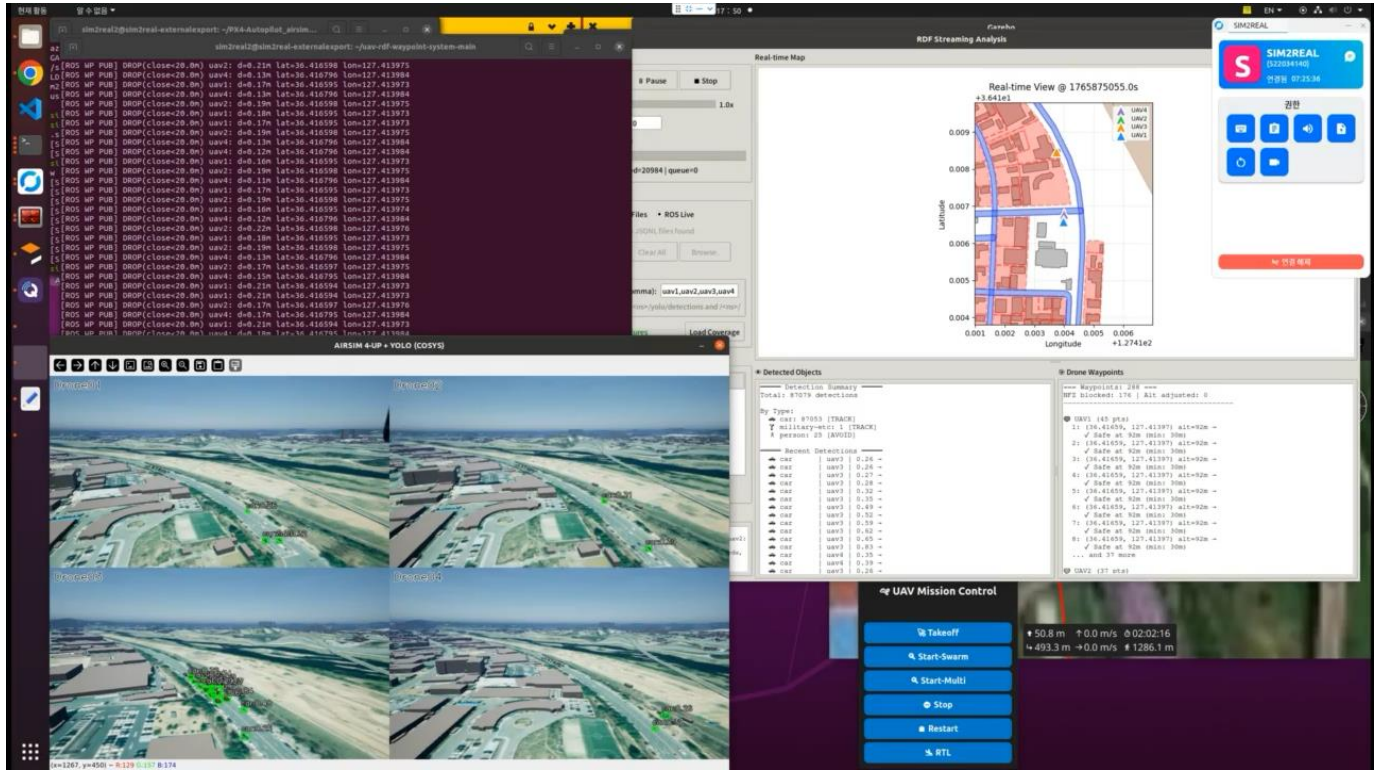
# 소개

한준구(코난쌤)
(현) 국립대학교 AI 강사 / 온톨로지 기반 에이전틱 AI 시스템 연구자
(현) NVIDIA DLI Ambassador

SK DEVOCEAN 온톨로지 & 지식그래프 스터디 참여
온톨로지 기반 자율 UAM 제어 에이전틱 시스템 구현
정부기관 AI 과제 공동연구원
AI 자동화 기술서 저자

# 참여했던 프로젝트

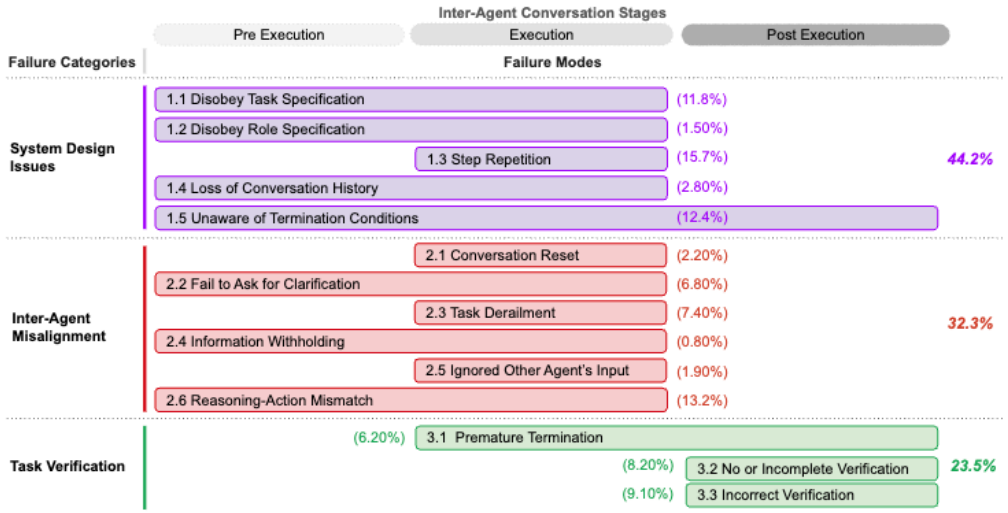
온톨로지 기반 UAV 제어
- 작전 상황 자동 판단 및 실시간 제어 (온톨로지 기반 제어 부분 담당)



# "Why Do Multi-Agent LLM Systems Fail?"

저자: Mert Cemri 외 12명 (UC Berkeley 등)

**연구 내용:**




Multi-Agent LLM Systems(MAS)에 대한 관심이 높아지고 있지만,
실제로는 기대만큼 성능 향상이 나타나지 않음.

# 원인 분석

| 대분류 | 세부 실패 모드 | 설명 |
|---|---|---|
| **1. System Design Issues (시스템 설계 문제)** | 1.1 Disobey Task Specification (작업 사양 미준수) | 주어진 작업의 제약 조건이나 요구 사항을 준수하지 못하여 최적이지 않거나 부정확한 결과 발생 |
| | 1.2 Disobey Role Specification (역할 사양 미준수) | 할당된 역할의 정의된 책임과 제약 조건을 준수하지 못하여 에이전트가 다른 에이전트처럼 행동 |
| | 1.4 Loss of Conversation History (대화 기록 손실) | 예상치 못한 문맥 잘림으로 인해 최근 상호작용 기록을 무시하고 이전 대화 상태로 회귀 |
| | 1.5 Unaware of Termination Conditions (종료 조건 인지 부족) | 에이전트 상호작용의 종료를 유발해야 하는 기준을 인식하지 못하여 불필요한 지속 |
| 2. Inter-Agent Misalignment (에이전트 간 불일치) | 2.2 Fail to Ask for Clarification (명확화 요청 실패) | 불분명하거나 불완전한 데이터에 직면했을 때 추가 정보를 요청하지 못하여 잘못된 행동 초래 |
| | 2.4 Information Withholding (정보 보류) | 중요한 데이터나 통찰력을 다른 에이전트와 공유하거나 전달하지 못함 |
| | 2.6 Reasoning-Action Mismatch (추론-행동 불일치) | 논리적 추론 과정과 실제 행동 간의 불일치로 인해 예상치 못한 결과 발생 |
| 3. Task Verification (작업 검증) | 3.1 Premature Termination (시기상조 종료) | 작업이 완료되기 전에 조기 종료 |
| | 3.2 No or Incomplete Verification (검증 부족 또는 불완전) | 결과의 정확성과 완전성을 확인하는 메커니즘 부재 또는 불충분 |
| | 3.3 Incorrect Verification (부정확한 검증) | 잘못된 기준이나 방법으로 검증을 수행 |

# 실패의 3대 카테고리

## 시스템 설계 문제

- **부적절한 워크플로우 패턴 선택**
- 에이전트 역할 분배의 비효율성
- **과도하게 복잡한 아키텍처**

## 에이전트 간 정렬 문제

- 통신 프로토콜의 불일치
- 공유 컨텍스트 관리 실패
- 목표(goal)에 대한 이해 차이

## 작업 검증 문제

- 중간 결과물의 품질 검증 부재
- 최종 출력의 정확성 확인 실패
- 에러 핸들링 메커니즘 부족

# 기존 Multi-Agent 시스템의 한계

### 하드코딩된 워크플로우

- 에이전트 조합이 코드에 고정
- 새 패턴 추가 시 코드 수정 필요
- 유연성 부족

### 관계 표현의 어려움

- 에이전트 간 관계 명시 불가
- 워크플로우 간 연결 표현 한계
- 확장성 제한

### 검색의 비효율성

- 키워드 기반 단순 매칭
- 의미적 검색 불가
- 컨텍스트 이해 부족

# 워크플로우 정보를 그래프로 저장하고 불러와서 쓰자

헷갈려 하지 않도록 -> 에이전트, Tool, 컨텍스트 등등이 포함된 그래프 먼저 만들기

어떤 그래프? RDF / LPG

RDF 는 구현 어려움, 다른 그래프 연결하기 쉬움 = kg1+kg2
LPG 는 구현하기 쉬움, 다른 그래프 연결은 어려움

근데 요청을 어떻게 쿼리로 바꾸지? Vector를 중간에 껴 넣기

# 목차 (Contents)

Part 1

# RDF 기초 개념

Resource Description Framework

# RDF (Resource Description Framework)란?

RDF는 웹 상의 자원(Resource)을 기술하기 위한 W3C 표준 프레임워크입니다.
데이터를 '주어-술어-목적어' 형태의 트리플(Triple)로 표현합니다.

### 핵심 특징

그래프 기반 데이터 모델
URI로 자원 식별
분산 데이터 통합 용이
의미론적 추론 가능

### 활용 분야

지식 그래프 (Knowledge Graph)
시맨틱 웹 (Semantic Web)
데이터 연결 (Linked Data)
온톨로지 정의

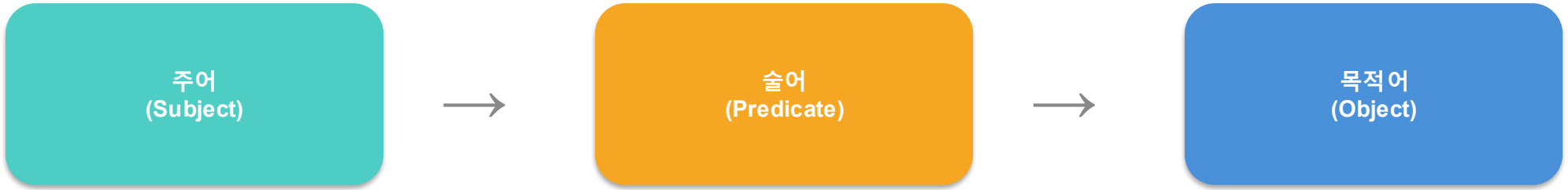
### 관련 기술

SPARQL - 쿼리 언어
OWL - 온톨로지 언어
Turtle - 직렬화 포맷
JSON-LD - 웹 친화적 포맷

**"지식을 그래프 형태로 표현하고 추론할 수 있는 표준"**

# 트리플 (Triple) - RDF의 기본 단위

RDF의 모든 데이터는 '주어(Subject) - 술어(Predicate) - 목적어(Object)' 형태의 트리플로 표현됩니다.




**예시:**

**RDF Triple Example**

```
# 자연어: "Web Research 도구는 data_acquisition 카테고리에 속한다"

# RDF 트리플:
tool:web_research  tool:category  "data_acquisition" .

#   주어: tool:web_research (도구 URI)
#   술어: tool:category (속성)
#   목적어: "data_acquisition" (리터럴 값)
```

# RDF 그래프 구조

여러 트리플이 모여 방향성 그래프(Directed Graph)를 형성합니다. 노드는 자원, 엣지는 관계를 나타냅니다.


## 그래프의 장점

관계 중심 데이터 모델링
데이터 통합 용이
경로 탐색 및 추론 가능

## 노드 유형

URI 노드: 고유 식별자를 가진 자원
리터럴 노드: 문자열, 숫자 등의 값
빈 노드(Blank Node): 익명 자원

# 네임스페이스 (Namespace)

네임스페이스는 URI를 축약하여 가독성을 높이고, 어휘의 출처를 명시합니다.

**Namespace Definition**

```
# 네임스페이스 정의 (Python rdflib)
from rdflib import Namespace

TOOL = Namespace("http://example.org/tool#")
WORKFLOW = Namespace("http://example.org/workflow#")

# 축약 전 (Full URI)
<http://example.org/tool#web_research> <http://example.org/tool#category> "data_acquisition" .

# 축약 후 (Prefixed)
tool:web_research  tool:category  "data_acquisition" .
```

**표준 네임스페이스:**

| Prefix | URI | 용도 |
|---|---|---|
| rdf: | http://www.w3.org/1999/02/22-rdf-syntax-ns# | RDF 기본 어휘 |
| rdfs: | http://www.w3.org/2000/01/rdf-schema# | RDF 스키마 |
| xsd: | http://www.w3.org/2001/XMLSchema# | 데이터 타입 |
| owl: | http://www.w3.org/2002/07/owl# | 온톨로지 |

# 온톨로지 (Ontology)

온톨로지는 특정 도메인의 개념과 관계를 정의한 '지식 체계'입니다. RDF 스키마(RDFS)나 OWL로 정의합니다.

### 온톨로지 구성요소

Class: 개념/타입 정의 (예: Tool, Workflow)
Property: 관계/속성 정의 (예: enables, category)
Individual: 실제 인스턴스 (예: web_research)
Constraint: 제약조건 (예: 도메인, 범위)

### 온톨로지 활용

데이터 일관성 검증
암묵적 지식 추론
시스템 간 상호운용성
쿼리 최적화

### Tool Ontology Definition

```
# 도구 온톨로지 정의 예시

# 클래스 정의
tool:Tool  rdf:type  rdfs:Class .

# 속성 정의
tool:category  rdf:type  rdf:Property ;
               rdfs:domain  tool:Tool ;
               rdfs:range   xsd:string .

# 관계 정의
workflow:enables  rdf:type  rdf:Property ;
                  rdfs:domain  tool:Tool ;
                  rdfs:range   tool:Tool .
```

Tool은 Class야.

category는 Property야, 주체(domain)는 Tool이고, 값(range)은 문자열이야.

enables는 Property야, 주체(domain)는 Tool이고, 대상(range)도 Tool이야.

# Turtle 직렬화 포맷

Turtle(Terse RDF Triple Language)은 RDF를 텍스트로 저장하는 가장 읽기 쉬운 포맷입니다.

**tool_workflow.ttl**

```
@prefix tool: <http://example.org/tool#> .
@prefix workflow: <http://example.org/workflow#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

tool:web_research
    a tool:Tool ;                          # rdf:type의 축약
    rdfs:label "Web Research" ;            # 이름
    rdfs:comment "웹 검색 및 정보 수집" ;    # 설명
    tool:category "data_acquisition" ;     # 카테고리
    tool:inputType "query" ;               # 입력 타입
    tool:outputType "search_results" ;     # 출력 타입
    workflow:enables tool:analyze_data .   # 후행 도구
```

tool은 http://example.org/tool#로 선언할게, workflow는 http://example.org/workflow#로 선언할게,
rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게.

web_research는:
Tool 타입이야 (a는 rdf:type 축약이야).
label은 "Web Research"야 (이름).
comment는 "웹 검색 및 정보 수집"이야 (설명).
category는 "data_acquisition"이야 (카테고리).
inputType은 "query"야 (입력 타입).
outputType은 "search_results"야 (출력 타입).
enables 관계로 analyze_data를 가능하게 해 (후행 도구).

## Turtle 문법

@prefix - 네임스페이스 선언
a - rdf:type 축약
; - 같은 주어에 대해 다른 술어-목적어 추가
, - 같은 주어-술어에 대해 다른 목적어 추가
. - 트리플 종료

## 기타 직렬화 포맷

RDF/XML - 표준 XML 기반 (장황함)
N-Triples - 한 줄에 하나의 트리플
JSON-LD - JSON 기반 (웹 API 친화적)
N-Quads - 그래프 이름 포함

# Python rdflib 라이브러리

rdflib은 Python에서 RDF를 다루기 위한 표준 라이브러리입니다.

**rdflib Basic Usage**

```
from rdflib import Graph, Dataset, Literal, Namespace, URIRef
from rdflib.namespace import RDF, RDFS

# 네임스페이스 정의
TOOL = Namespace("http://example.org/tool#")
WORKFLOW = Namespace("http://example.org/workflow#")

# 그래프 생성
graph = Dataset(default_union=True)

# 트리플 추가
tool_uri = TOOL["web_research"]
graph.add((tool_uri, RDF.type, TOOL.Tool))
graph.add((tool_uri, RDFS.label, Literal("Web Research")))
graph.add((tool_uri, TOOL.category, Literal("data_acquisition")))

# 그래프 직렬화
graph.serialize("tools.ttl", format="turtle")

print(f"트리플 수: {len(graph)}")
```

rdflib에서 Graph, Dataset, Literal, Namespace, URIRef 가져와.
RDF, RDFS 네임스페이스도 가져와.

TOOL은 http://example.org/tool#로 만들어, WORKFLOW는
http://example.org/workflow#로 만들어.

그래프 만들어 (Dataset으로, default_union은 True로).

tool_uri는 TOOL["web_research"]로 만들어.
그래프에 추가해: tool_uri는 Tool 타입이야.
그래프에 추가해: tool_uri의 label은 "Web Research"야.
그래프에 추가해: tool_uri의 category는 "data_acquisition"이야.

그래프를 "tools.ttl" 파일로 저장해 (turtle 포맷으로).
"트리플 수: 몇 개" 출력해.

# Part 1 요약: RDF 핵심 개념

| | |
|---|---|
| **Triple** | 주어-술어-목적어 형태의 데이터 단위 |
| **Graph** | 트리플들이 모여 형성하는 방향성 그래프 |
| **Namespace** | URI를 축약하여 가독성 향상 |
| **Ontology** | 도메인 개념과 관계의 정의 |
| **Turtle** | RDF의 텍스트 직렬화 포맷 |

**RDF = 지식을 그래프로 표현하고 추론하는 표준**

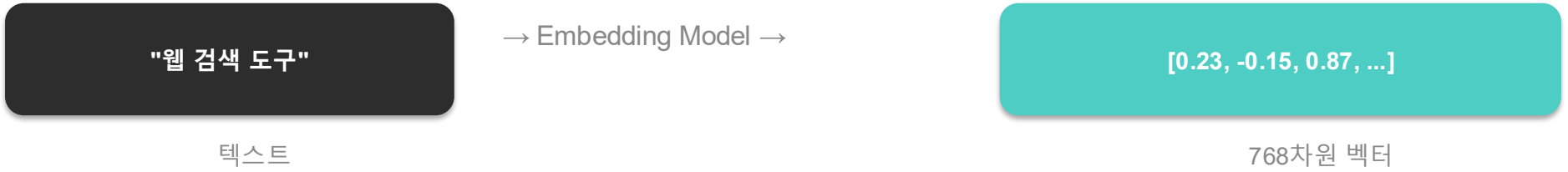
Part 2

# Vector Embedding 기초

텍스트를 수치 벡터로 변환하여 의미 검색 수행

# Vector Embedding이란?

임베딩(Embedding)은 텍스트, 이미지 등의 데이터를 고정 차원의 실수 벡터로 변환하는 기술입니다.
의미적으로 유사한 데이터는 벡터 공간에서 가까운 위치에 매핑됩니다.




## 임베딩의 특성

고정 차원 출력 (예: 768, 1536차원)
의미적 유사성 보존
벡터 연산으로 유사도 계산 가능
다양한 데이터 타입 지원

## 주요 임베딩 모델

OpenAI text-embedding-3-large (3072d)
ko-sroberta-multitask (768d) - 한국어
all-MiniLM-L6-v2 (384d) - 경량
BERT, RoBERTa 계열

# 벡터 유사도 계산

두 벡터 간의 거리/유사도를 계산하여 의미적 유사성을 측정합니다.

### Similarity Metrics

```
# 코사인 유사도 (Cosine Similarity)
import numpy as np

def cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) *
np.linalg.norm(v2))

# 결과: -1 ~ 1 (1에 가까울수록 유사)

# L2 거리 (Euclidean Distance)
def l2_distance(v1, v2):
    return np.linalg.norm(v1 - v2)

# 결과: 0 ~ inf (0에 가까울수록 유사)
```

### Cosine Similarity

방향 기반 유사도
벡터 크기(magnitude) 무시
텍스트 유사도에 주로 사용
범위: -1 ~ 1

### L2 Distance

유클리드 거리
벡터 크기 고려
FAISS IndexFlatL2 사용
범위: 0 ~ inf

**FAISS는 대규모 벡터에서 근사 최근접 이웃(ANN)을 빠르게 검색**

# FAISS (Facebook AI Similarity Search)

FAISS는 Meta에서 개발한 고성능 벡터 유사도 검색 라이브러리입니다.

### FAISS Usage Example

```
from faiss import IndexFlatL2
import numpy as np

# 1. 벡터 준비 (N개 벡터, D차원)
embeddings = np.array(embedding_list).astype("float32")  # shape: (N, D)

# 2. 인덱스 생성 (L2 거리 기반)
dimension = embeddings.shape[1]  # 예: 768
index = IndexFlatL2(dimension)

# 3. 벡터 추가
index.add(embeddings)
print(f"인덱스된 벡터 수: {index.ntotal}")

# 4. 검색 (Top-K)
query_vector = np.array([query_embedding]).astype("float32")
distances, indices = index.search(query_vector, k=5)

# distances: 거리 값 배열, indices: 벡터 인덱스 배열
```

faiss에서 IndexFlatL2 가져와, numpy도 가져와.

벡터 준비해: embedding_list를 numpy 배열로 바꿔서 float32로 변환해
(shape는 N개 벡터, D차원).

인덱스 만들어 (L2 거리 기반으로):
dimension은 embeddings의 열 개수야 (예: 768).
**IndexFlatL2**로 dimension 크기만큼 인덱스 만들어.

벡터 추가해: embeddings를 index에 넣어.
"인덱스된 벡터 수: 몇 개" 출력해.

검색해 (Top-K로):
query_vector 만들어 (query_embedding을 배열로 바꿔서 float32로).
index에서 검색해 (k=5로, 가까운 5개 찾아).
distances는 거리 값 배열이고, indices는 벡터 인덱스 배열이야.

### FAISS 인덱스 유형

**IndexFlatL2 - 정확한 검색 (Brute Force)**
IndexIVFFlat - 클러스터 기반 근사
IndexHNSW - 그래프 기반 근사
IndexPQ - 양자화 기반 압축

# Sentence Transformers (한국어 임베딩)

Sentence Transformers는 문장 수준의 임베딩을 생성하는 라이브러리입니다.
한국어에는 ko-sroberta-multitask 모델을 사용합니다.

### Sentence Transformers Usage

```
from sentence_transformers import SentenceTransformer

# 한국어 임베딩 모델 로드
model = SentenceTransformer("jhgan/ko-sroberta-multitask")

# 단일 문장 임베딩
embedding = model.encode("웹 검색 및 정보 수집 도구")
print(f"차원: {len(embedding)}")  # 768

# 여러 문장 임베딩 (배치)
sentences = [
    "웹 검색 및 정보 수집",
    "데이터 분석 및 통계",
    "시각화 차트 생성"
]
embeddings = model.encode(sentences, normalize_embeddings=True)
print(f"Shape: {embeddings.shape}")  # (3, 768)
```

SentenceTransformer 가져와.

한국어 임베딩 모델 로드해 (jhgan/ko-sroberta-multitask로).

단일 문장 임베딩 만들어: "웹 검색 및 정보 수집 도구"를 encode해.
"차원: 몇 차원" 출력해 (768이야).

여러 문장 임베딩 만들어 (배치로):
sentences 리스트 만들어
("웹 검색 및 정보 수집", "데이터 분석 및 통계", "시각화 차트 생성").
sentences를 encode해 (normalize_embeddings는 True로).
"Shape: 모양" 출력해 ((3, 768)이야).

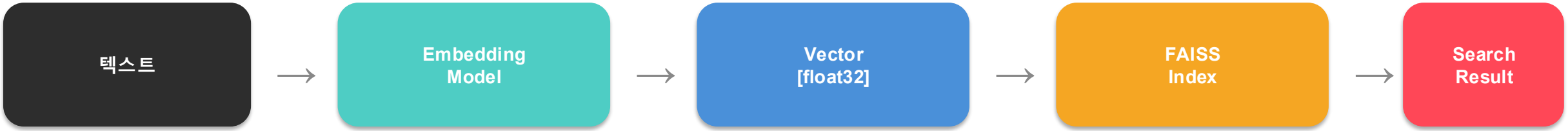
**ko-sroberta-multitask**

한국어 특화 모델
RoBERTa 기반
768 차원 출력
문장 유사도에 최적화

**LangChain 호환 래퍼**

embed_query(text) - 단일
embed_documents(texts) - 배치
normalize_embeddings=True

# Part 2 요약: Vector Embedding




## 핵심 개념

Embedding: 텍스트 → 고차원 실수 벡터
**의미적 유사성이 벡터 거리로 반영됨**
Cosine/L2로 유사도 측정

## 기술 스택

SentenceTransformers - 임베딩 생성
FAISS - 대규모 벡터 검색
ko-sroberta - 한국어 모델 (768d)

**Vector Search = 의미 기반 검색 (키워드 매칭 X)**

Part 3

# RDF → Vector 변환

지식 그래프를 벡터 공간으로 임베딩

# 왜 RDF 그래프를 임베딩하는가?

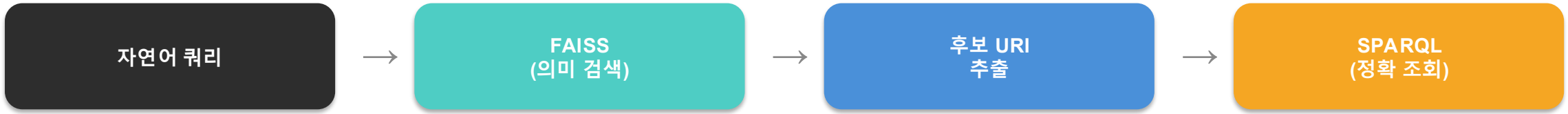
**RDF만의 한계**

SPARQL 쿼리는 정확한 매칭만 가능
자연어 쿼리 처리 어려움
오타, 유의어 처리 불가
대규모 그래프에서 검색 속도 저하

**Vector Search의 장점**

의미 기반 유사도 검색
자연어 쿼리 지원
오타, 유의어 자동 처리
밀리초 단위 검색 속도

## RDF + Vector = 하이브리드 검색




1. Vector Search로 관련 노드 빠르게 찾기
2. RDF Graph에서 연결된 정보 SPARQL로 조회
3. 두 기술의 장점 결합

# Step 1: RDF 노드를 텍스트로 변환

RDF 노드의 속성들을 하나의 텍스트 문자열로 결합합니다. 이 텍스트가 임베딩 입력이 됩니다.

**SPARQL Query for Embedding Text**

```
# SPARQL로 임베딩용 텍스트 추출
q_tool_embedding = """
PREFIX tool: <http://example.org/tool#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?uri ?label ?description ?category
WHERE {
    ?uri a tool:Tool ;
        rdfs:label ?label ;
        rdfs:comment ?description ;
        tool:category ?category .
}
"""

# 결과를 텍스트로 변환
for row in graph.query(q_tool_embedding):
    embedding_text = f"tool: {row.label}; description: {row.description}; category:
{row.category}"
    # 예: "tool: Web Research; description: 웹 검색 및 정보 수집; category:
data_acquisition"
```

tool은 http://example.org/tool#로 선언할게,
rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게.

uri 찾아, label 찾아, description 찾아, category 찾아.

근데 조건이 있어.
Tool 타입이어야 하고, label 속성 있어야 하고, description(comment) 속성 있어야 하고, category 속성도 있어야 해.

쿼리 결과를 하나씩 돌려.
각 row마다 embedding_text 만들어,
"tool: 이름; description: 설명; category: 카테고리" 이런 형식으로.

예를 들면 "tool: Web Research; description: 웹 검색 및 정보 수집; category: data_acquisition" 이런 식으로 나와.

**핵심: 노드의 핵심 속성(label, description, category)을 "key: value; ..." 형식으로 결합**

# Step 2: 텍스트를 벡터로 변환

### Embedding Generation Loop

```
embeddings = []       # 벡터 리스트
uri_list = []         # URI 리스트 (인덱스 매핑용)
metadata_list = []    # 메타데이터 리스트

for row in graph.query(q_tool_embedding):
    # 1. 임베딩 텍스트 생성
    embedding_text = f"tool: {row.label}; description:
{row.description}; category: {row.category}"

    # 2. 벡터 생성 (768차원)
    embedding = embedding_model.embed_query(embedding_text)

    # 3. 리스트에 저장
    embeddings.append(embedding)
    uri_list.append(str(row.uri))  # URI로 RDF 노드 식별
    metadata_list.append({
        "label": str(row.label),
        "description": str(row.description),
        "category": str(row.category)
    })

print(f"총 {len(embeddings)}개의 도구 임베딩 생성")
```

embeddings 빈 리스트 만들어 (벡터 담을 거야).
uri_list 빈 리스트 만들어 (URI 담을 거야, 인덱스 매핑용).
metadata_list 빈 리스트 만들어 (메타데이터 담을 거야).

q_tool_embedding 쿼리 돌려서 각 row마다:
embedding_text 만들어 "tool: 이름; description: 설명; category: 카테고리" 형식으로.

벡터 만들어 (embedding_model로 embedding_text를 embed_query 해, 768차원으로).

embeddings에 벡터 추가해.
uri_list에 URI 문자열로 변환해서 추가해 (RDF 노드 식별용).
metadata_list에 딕셔너리 추가해 (label, description, category 담아서).

"총 몇 개의 도구 임베딩 생성" 출력해.

### 저장되는 데이터

embeddings[i] - 768차원 벡터
uri_list[i] - RDF 노드 URI
metadata_list[i] - 추가 정보

인덱스(i)로 세 리스트가 연결됨

# Step 3: FAISS 인덱스 구축

### FAISS Index Creation

```
import numpy as np
from faiss import IndexFlatL2

# 1. numpy 배열로 변환 (FAISS는 float32 요구)
xb = np.array(embeddings).astype("float32")
print(f"Shape: {xb.shape}")

# 2. L2 거리 기반 인덱스 생성
dimension = xb.shape[1]  # 768
index = IndexFlatL2(dimension)

# 3. 벡터 추가
index.add(xb)

print(f"FAISS 인덱스 생성 완료!")
print(f"  차원: {dimension}")
print(f"  벡터 수: {index.ntotal}")
```

numpy 가져와, faiss에서 IndexFlatL2 가져와.

embeddings를 numpy 배열로 바꿔서 float32로 변환해 (FAISS는 float32 필요해).
"Shape: 모양" 출력해 ((8, 768)이야 - 8개 도구, 768차원).

L2 거리 기반 인덱스 만들어:
dimension은 xb의 열 개수야 (768이야).
IndexFlatL2로 dimension 크기만큼 index 만들어.

벡터 추가해: xb를 index에 넣어.
"FAISS 인덱스 생성 완료!" 출력해.
" 차원: 몇 차원" 출력해.
" 벡터 수: 몇 개" 출력해.

### IndexFlatL2 특성

정확한 Brute Force 검색
소규모 데이터셋에 적합 (~1M)
추가 학습 불필요
메모리: O(N * D * 4 bytes)

**인덱스 구조:**

| Index | URI | Vector (768d) |
|---|---|---|
| 0 | tool:web_research | [0.23, -0.15, 0.87, ...] |
| 1 | tool:fetch_url | [0.45, 0.32, -0.21, ...] |
| 2 | tool:analyze_data | [-0.12, 0.67, 0.44, ...] |
| ... | ... | ... |

# Step 4: 검색 함수 구현

### search_tools Function

```
def search_tools(query: str, top_k: int = 3) -> list:
    """
    자연어 쿼리로 관련 도구 검색
    """
    # 1. 쿼리 텍스트를 벡터로 변환
    x_query = np.array([embedding_model.embed_query(query)]).astype("float32")

    # 2. FAISS 검색 (Top-K)
    distances, indices = index.search(x_query, top_k)

    # 3. 결과 매핑
    results = []
    for i, idx in enumerate(indices[0]):
        if idx < len(uri_list):
            results.append({
                "uri": uri_list[idx],          # RDF 노드 URI
                "metadata": metadata_list[idx], # 저장된 메타데이터
                "score": float(distances[0][i])  # L2 거리 (낮을수록 유사)
            })
    return results
```

search_tools 함수 만들어: query랑 top_k 받아서 (기본값 3)
관련 도구 검색해서 리스트로 반환해.

쿼리 텍스트를 벡터로 바꿔:
embedding_model로 query를 embed_query 해서 배열로
만들고 float32로 변환해.

FAISS 검색해 (Top-K로):
index에서 x_query 검색해 (top_k개 찾아).
distances랑 indices 받아.

결과 매핑해:
results 빈 리스트 만들어.
indices[0]의 각 idx마다 (인덱스 i도 같이):
idx가 uri_list 길이보다 작으면:
results에 딕셔너리 추가해 (uri는 uri_list[idx],
metadata는 metadata_list[idx], score는 distances[0][i]를
float로 - L2 거리야, 낮을수록 유사해).

results 반환해.

### Usage Example

```
# 사용 예시
results = search_tools("웹 페이지 분석", top_k=2)

# 결과:
# [{"uri": "tool:web_research", "metadata": {...}, "score": 0.45},
#  {"uri": "tool:fetch_url", "metadata": {...}, "score": 0.62}]
```

Part 4

# SPARQL 쿼리

RDF 그래프에서 정보 추출하기

# SPARQL 기본 문법

SPARQL은 RDF 데이터를 조회하기 위한 W3C 표준 쿼리 언어입니다.

### SPARQL Query Structure

```
# SPARQL 쿼리 구조

PREFIX tool: <http://example.org/tool#>     # 1. 네임스페이스 선언
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?description                  # 2. 반환할 변수
WHERE {                                     # 3. 그래프 패턴
    ?tool a tool:Tool .                     #    타입이 Tool인 것
    ?tool rdfs:label ?label .               #    레이블 추출
    ?tool rdfs:comment ?description .       #    설명 추출
}
ORDER BY ?label                             # 4. 정렬
LIMIT 10                                    # 5. 개수 제한
```

tool은 http://example.org/tool#로 선언할게,
rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게.

label 찾아, description 찾아.

근데 조건이 있어.
tool이 Tool 타입이어야 하고, 그 tool의 label 속성 있어야 하고,
comment(description) 속성도 있어야 해.

label 기준으로 정렬해, 10개만 가져와.

### 핵심 키워드

PREFIX - 네임스페이스 축약
SELECT - 반환할 변수 지정
WHERE - 트리플 패턴 매칭
OPTIONAL - 선택적 패턴
FILTER - 조건 필터링

### 패턴 문법

?var - 변수 (물음표로 시작)
a - rdf:type 축약
. - 트리플 종료
; - 같은 주어, 다른 술어

# SPARQL 예제

## Query 1: Tool Dependencies

```
# 예제 1: 모든 도구의 선후관계 조회
query_deps = """
PREFIX tool: <http://example.org/tool#>
PREFIX workflow: <http://example.org/workflow#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?tool ?label ?requires ?enables
WHERE {
    ?tool a tool:Tool .
    ?tool rdfs:label ?label .
    OPTIONAL { ?tool workflow:requires ?reqTool . ?reqTool
rdfs:label ?requires }
    OPTIONAL { ?tool workflow:enables ?enTool . ?enTool rdfs:label
?enables }
}
ORDER BY ?label
"""
```

tool은 http://example.org/tool#로 선언할게,
workflow는 http://example.org/workflow#로 선언할게,
rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게. tool 찾아, label 찾아, requires 찾아, enables 찾아.
근데 조건이 있어.
tool이 Tool 타입이어야 하고, label 속성 있어야 해. 있으면 가져오는 거: requires 관계 있으면 그 도구 이름 가져와, enables 관계 있으면 그 도구 이름 가져와. label 기준으로 정렬해.

## Query 2: Tool Detail

```
# 예제 2: 특정 도구의 상세 정보
query_detail = """
PREFIX tool: <http://example.org/tool#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label ?desc ?input ?output
WHERE {
    tool:web_research
        rdfs:label ?label ;
        rdfs:comment ?desc ;
        tool:inputType ?input ;
        tool:outputType ?output .
}
"""
```

tool은 http://example.org/tool#로 선언할게, rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게. label 찾아, desc 찾아, input 찾아, output 찾아. 근데 web_research 도구의 정보만 가져와. label 속성, comment(desc) 속성, inputType 속성, outputType 속성 다 있어야 해.

## Execute in Python

```
# Python에서 실행
results = list(graph.query(query_deps))
for row in results:
    print(f"Tool: {row.label}, Requires: {row.requires}, Enables: {row.enables}")
```

쿼리 돌려서 결과를 리스트로 만들어.
각 row마다 "Tool: 이름, Requires: 필요한거, Enables: 가능하게하는거"
이런 식으로 출력해.

# SPARQL 고급 패턴

## FILTER & BIND

```
# 1. FILTER - 조건 필터링
SELECT ?tool ?label WHERE {
    ?tool a tool:Tool ;
          rdfs:label ?label ;
          tool:category ?cat .
    FILTER (?cat = "data_acquisition")    # 특정 카테고리만
}
```

tool 찾아, label 찾아. Tool 타입이어야 하고, label 속성 있어야 하고, category 속성도 있어야 해. 근데 category가 "data_acquisition"인 것만 걸러.

```
# 2. BIND - 변수 생성
SELECT ?tool ?fullName WHERE {
    ?tool rdfs:label ?label .
    BIND (CONCAT("[Tool] ", ?label) AS ?fullName)
}
```

tool 찾아, fullName 찾아.
label 속성 있어야 해.
fullName은 "[Tool] "이랑 label을 합쳐서 만들어.

## Path & UNION

```
# 3. 경로 패턴 - 여러 홉 탐색
SELECT ?start ?end WHERE {
    ?start workflow:enables+ ?end .  # + = 1개 이상
    # * = 0개 이상, ? = 0 또는 1개
}
```

start 찾아, end 찾아.
start에서 enables 관계를 한 번 이상 따라가서 end까지 도달할 수 있는 것들 찾아.
참고로 +는 1개 이상, *는 0개 이상, ?는 0 또는 1개야.

```
# 4. UNION - OR 조건
SELECT ?tool WHERE {
    { ?tool workflow:requires tool:web_research }
    UNION
    { ?tool workflow:requires tool:fetch_url }
}
```

tool 찾아.
web_research를 필요로 하는 tool이거나, fetch_url을 필요로 하는 tool이거나 둘 중 하나면 돼.

# 실제 구현: get_tool_dependencies()

**get_tool_dependencies Implementation**

```
def get_tool_dependencies(tool_name: str) -> dict:
    """
    RDF 그래프에서 도구의 선후관계 조회
    """
    query = f"""
    PREFIX tool: <http://example.org/tool#>
    PREFIX workflow: <http://example.org/workflow#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

    SELECT ?label ?reqLabel ?enLabel
    WHERE {{
        tool:{tool_name} rdfs:label ?label .
        OPTIONAL {{ tool:{tool_name} workflow:requires ?req . ?req rdfs:label ?reqLabel }}
        OPTIONAL {{ tool:{tool_name} workflow:enables ?en . ?en rdfs:label ?enLabel }}
    }}
    """

    results = list(graph.query(query))

    deps = {"tool": tool_name, "label": "", "requires": [], "enables": []}
    for row in results:
        deps["label"] = str(row.label) if row.label else tool_name
        if row.reqLabel and str(row.reqLabel) not in deps["requires"]:
            deps["requires"].append(str(row.reqLabel))
        if row.enLabel and str(row.enLabel) not in deps["enables"]:
            deps["enables"].append(str(row.enLabel))

    return deps
```

함수 만들어: tool_name 받아서 그 도구의 선후관계 dict로 반환해.

**쿼리 만들어:**
**tool은 http://example.org/tool#로 선언할게, workflow는 http://example.org/workflow#로 선언할게, rdfs는 http://www.w3.org/2000/01/rdf-schema#로 선언할게.**

**label 찾아, reqLabel 찾아, enLabel 찾아.**

**tool_name의 label 속성 있어야 해.**
**있으면 가져오는 거: requires 관계 있으면 그 도구 이름 reqLabel로 가져와, enables 관계 있으면 그 도구 이름 enLabel로 가져와.**

쿼리 돌려서 결과를 리스트로 만들어.

deps 딕셔너리 만들어: tool은 tool_name, label은 빈 문자열, requires는 빈 리스트, enables는 빈 리스트.

각 row마다 돌면서:
label 있으면 넣어, 없으면 tool_name 넣어.
reqLabel 있고 중복 아니면 requires 리스트에 추가해.
enLabel 있고 중복 아니면 enables 리스트에 추가해.

deps 반환해.

# 전체 직렬화 워크플로우

**Complete Serialization Workflow**

```
# 1. 그래프 구축
graph = Graph()
graph.bind("tool", TOOL)
graph.bind("workflow", WORKFLOW)
# ... 트리플 추가 ...

# 2. SPARQL로 특정 데이터 추출 후 새 그래프 생성
filtered_graph = graph.query("""
    CONSTRUCT { ?s ?p ?o }
    WHERE { ?s a tool:Tool ; ?p ?o .
            FILTER (?p != workflow:requires) }
""").graph

# 3. 다양한 포맷으로 직렬화
filtered_graph.serialize("tools_filtered.ttl", format="turtle")
filtered_graph.serialize("tools_filtered.jsonld", format="json-ld")

# 4. 원본 그래프 저장
graph.serialize("full_graph.ttl", format="turtle")
print(f"원본: {len(graph)} 트리플, 필터링: {len(filtered_graph)} 트리플")
```

그래프 만들어.
tool은 TOOL로 바인딩해, workflow는 WORKFLOW로 바인딩해.
트리플 추가해 (생략).

SPARQL로 필터링된 그래프 만들어:
쿼리: s, p, o를 그대로 가져와서 새 그래프 만들어 (CONSTRUCT).
조건: s가 Tool 타입이고, s의 속성 p랑 값 o 가져와.
근데 p가 workflow:requires인 건 빼.
.graph로 결과 그래프 받아서 filtered_graph에 저장해.

filtered_graph를 "tools_filtered.ttl" 파일로 저장해 (turtle 포맷으로).
filtered_graph를 "tools_filtered.jsonld" 파일로 저장해 (json-ld 포맷으로).

원본 graph를 "full_graph.ttl" 파일로 저장해 (turtle 포맷으로).

"원본: 몇 개 트리플, 필터링: 몇 개 트리플" 출력해.

Part 5

# 시스템 아키텍처

전체 시스템 구조와 데이터 흐름

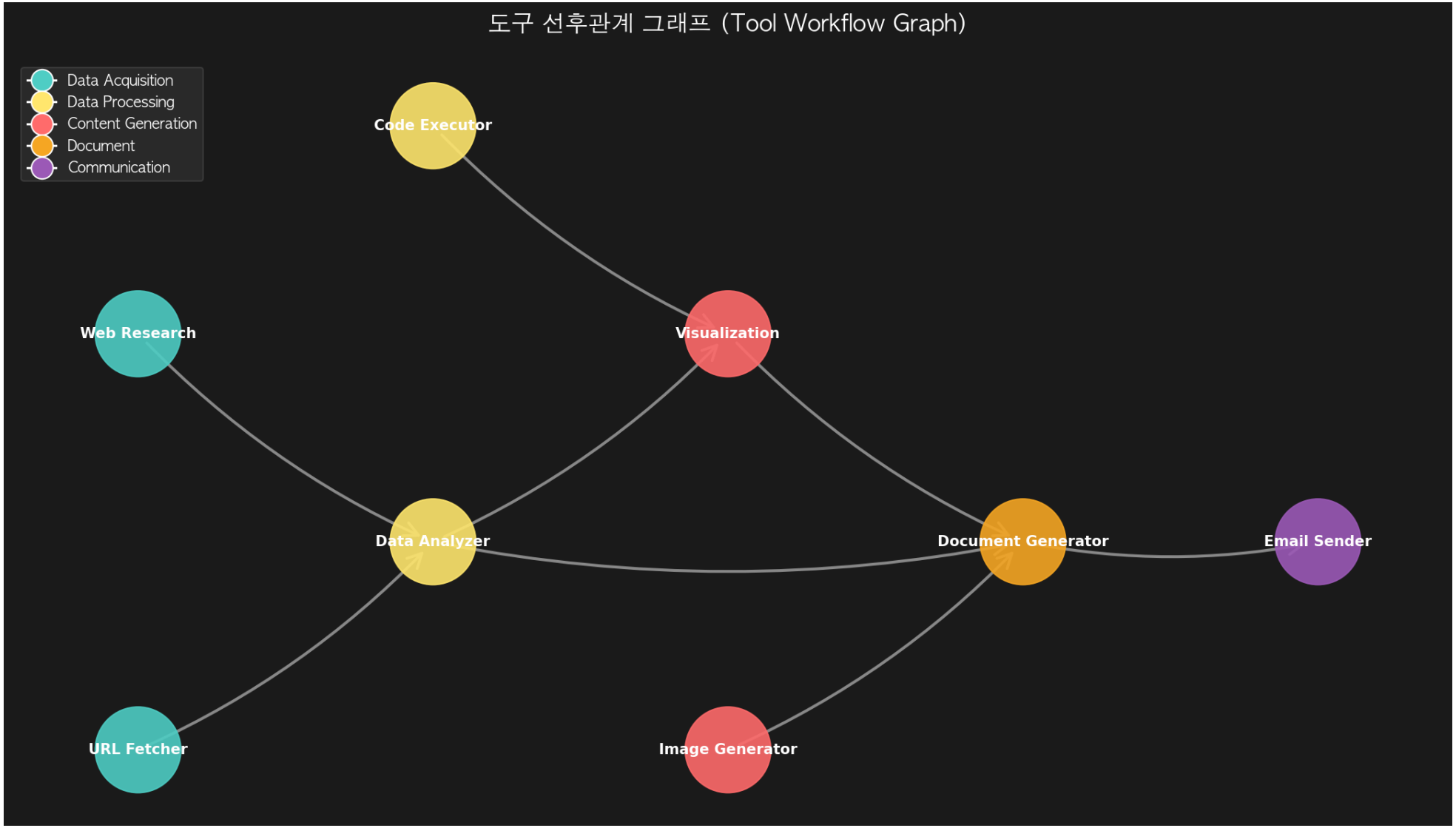
# 전체 시스템 아키텍처


사용자 (자연어 쿼리)
LangChain Agent
(tool-calling)
FAISS Index
(의미 검색)
RDF Graph
(지식 그래프)
관련 도구
URI 반환
선후관계
조회 (SPARQL)
도구 실행 (8개 도구)

# 처리 흐름 (Pipeline)

| | | |
|---|---|---|
| 1 | **사용자 쿼리** | "Python 트렌드 분석해서 문서로 만들어줘" |
| 2 | **FAISS 검색** | web_research, analyze_data, generate_document 발견 |
| 3 | **SPARQL 조회** | 선후관계: web → analyze → visualize → document |
| 4 | **워크플로우 생성** | WorkflowContext에 의존성 순서 저장 |
| 5 | **도구 실행** | 각 도구 순차 실행, 결과 컨텍스트 저장 |
| 6 | **결과 반환** | 생성된 파일 경로 및 요약 반환 |

# 도구 선후관계 그래프


도구 선후관계 그래프 (Tool Workflow Graph)
Data Acquisition
Data Processing
Content Generation
Document
Communication
Code Executor
Web Research
Visualization
Data Analyzer
Document Generator
Email Sender
URL Fetcher
Image Generator

# 하이브리드 검색 (Vector + SPARQL)

**Hybrid Search**

```
class HybridSearchEngine:
    async def discover_workflows(self, query: str, top_k: int = 5):
        # 1. 벡터 검색 (의미적 유사도)
        query_embedding = await self.embedder.embed(query)
        vector_results = self.faiss_index.search(query_embedding, top_k * 2)

        # 2. SPARQL 키워드 검색
        keywords = self._extract_keywords(query)
        sparql_results = self._keyword_search(keywords)

        # 3. RRF (Reciprocal Rank Fusion) 병합
        merged = self._rrf_merge(vector_results, sparql_results, k=60)

        # 4. SPARQL로 상세 정보 조회
        enriched = []
        for uri, score in merged[:top_k]:
            details = self._get_workflow_details(uri)  # SPARQL
            enriched.append(WorkflowCandidate(
                uri=uri, score=score, **details
            ))

        return enriched
```

query 받아서 워크플로우 찾아.

1. 벡터 검색해:
   query를 임베딩으로 바꿔.
   FAISS에서 유사한 거 top_k*2개 찾아.

2. SPARQL 키워드 검색해:
   query에서 키워드 뽑아.
   RDF 그래프에서 키워드 매칭해.

3. RRF로 합쳐:
   벡터 결과랑 SPARQL 결과 순위 합쳐.
   score = Σ 1/(60 + rank)

4. 상세 정보 가져와:
   합친 결과 top_k개의 URI로
   SPARQL 상세 조회해.

## RRF 공식

- score = Σ 1/(k + rank)
- k = 60 (하이퍼파라미터)
- 두 검색 결과 순위 결합
- 상위 결과 더 가중치

## 검색 특성

- Vector: 의미적 유사도
- SPARQL: 정확한 키워드
- RRF: 두 장점 결합
- 상호 보완적 결과

# Ko-SBERT 한국어 임베딩

### Ko-SBERT 임베딩

```
# Ko-SBERT 모델 로드 (한국어 특화)
from sentence_transformers import SentenceTransformer
import numpy as np

# 모델: jhgan/ko-sroberta-multitask
# 768차원 한국어 Sentence-BERT
EMBEDDING_MODEL = "jhgan/ko-sroberta-multitask"
embedder = SentenceTransformer(EMBEDDING_MODEL)

def get_embedding(text: str) -> np.ndarray:
    """텍스트를 Ko-SBERT 임베딩으로 변환"""
    return embedder.encode(text, convert_to_numpy=True).astype(np.float32)

# 테스트
emb = get_embedding("AI 트렌드 조사")
print(f"Shape: {emb.shape}")  # (768,)
```

Ko-SBERT 모델 로드해.
jhgan/ko-sroberta-multitask 모델이야.
한국어 특화된 768차원 임베딩 만들어줘.

get_embedding 함수 만들어:
text 받아서 numpy 벡터로 변환해.
float32로 캐스팅해 (FAISS용).

테스트해봐:
"AI 트렌드 조사" → (768,) 벡터 나와.

### Ko-SBERT 특징

- 한국어 의미 유사도 최적화
- 768차원 임베딩 벡터
- 로컬 실행 (API 불필요)
- 완전 무료

### vs Gemini 임베딩

- API 호출 불필요 → 빠름
- 한국어 특화 → 정확도↑
- Rate limit 없음
- 오프라인 사용 가능

# FAISS 벡터 인덱스 구축

### FAISS 인덱스 구축

```
import faiss

# FAISS 인덱스 생성 (Inner Product = 코사인 유사도)
EMBEDDING_DIM = 768
index = faiss.IndexFlatIP(EMBEDDING_DIM)
metadata_list = []

# 워크플로우마다 임베딩 생성
for wf in workflows:
    # 학습 문서: label + keywords + useCases
    text = f"{wf['label']} {wf['keywords']} {wf['useCases']}"

    emb = get_embedding(text)
    faiss.normalize_L2(emb.reshape(1, -1))  # 정규화
    index.add(emb.reshape(1, -1))
    metadata_list.append(wf)

print(f"FAISS 인덱스: {index.ntotal}개")
```

FAISS 인덱스 만들어.
IndexFlatIP는 Inner Product 기반이야.
정규화하면 코사인 유사도랑 같아.

워크플로우마다 돌면서:
1. 학습 문서 만들어:
   "{label} {keywords} {useCases}"

2. Ko-SBERT로 임베딩 만들어.

3. L2 정규화해.

4. 인덱스에 추가해.

5. 메타데이터도 저장해.

### 학습 문서 구조

- label: 워크플로우 이름
- keywords: 검색 키워드
- useCases: 사용 사례
- → "{label} {keywords} {useCases}"

### 학습 문서 예시

- "AI 트렌드 조사
- AI, 인공지능, 트렌드, 조사...
- 기술 동향 파악, 시장 조사"

### IndexFlatIP

- Inner Product 기반
- 정규화 후 = 코사인 유사도
- 정확한 검색 (근사 아님)

# 인덱스 저장/로드 & 벡터 검색

## 인덱스 저장/로드

```
import json

INDEX_PATH = "workflow_index.faiss"
METADATA_PATH = "workflow_metadata.json"

def save_index(idx, metadata):
    """FAISS 인덱스와 메타데이터 저장"""
    faiss.write_index(idx, INDEX_PATH)
    with open(METADATA_PATH, "w", encoding="utf-8") as f:
        json.dump(metadata, f, ensure_ascii=False, indent=2)

def load_index():
    """저장된 인덱스 로드"""
    idx = faiss.read_index(INDEX_PATH)
    with open(METADATA_PATH, "r", encoding="utf-8") as f:
        metadata = json.load(f)
    return idx, metadata
```

save_index 함수:
faiss.write_index()로 인덱스 저장해.
메타데이터는 JSON으로 저장해.

load_index 함수:
faiss.read_index()로 인덱스 로드해.
메타데이터도 JSON에서 읽어와.

## 저장 파일

- .faiss: 벡터 인덱스 (바이너리)
- .json: 메타데이터 (URI, label 등)
- 재시작 시 빠른 로드

## 벡터 검색 함수

```
def vector_search(query: str, top_k: int = 3):
    """Ko-SBERT + FAISS 벡터 검색"""
    q_vec = get_embedding(query)
    faiss.normalize_L2(q_vec.reshape(1, -1))

    scores, indices = index.search(
        q_vec.reshape(1, -1), top_k
    )

    results = []
    for score, idx in zip(scores[0], indices[0]):
        if idx >= 0:
            r = metadata_list[idx].copy()
            r["score"] = float(score)
            results.append(r)
    return results

# 검색 예시
results = vector_search("AI 최신 트렌드")
# → [{'label': 'AI 트렌드 조사', 'score': 0.89}]
```

vector_search 함수:
query 받아서 top_k개 찾아.

1. query를 임베딩으로 바꿔.
2. L2 정규화해.
3. FAISS에서 검색해.
4. 결과에 score 붙여서 반환해.

예시: "AI 최신 트렌드" 검색하면
→ [{'label': 'AI 트렌드 조사', 'score': 0.89}]

# LLM 기반 워크플로우 선택

**LLM Selection**

```
async def select_workflow_with_llm(query: str, candidates:
List[WorkflowCandidate]):
    prompt = f"""사용자 요청에 가장 적합한 워크플로우를 선택하세요.

## 사용자 요청
{query}

## 후보 워크플로우
{format_candidates(candidates)}

## 선택 기준
- 사용자 의도와 워크플로우 목적 일치도
- 에이전트 구성의 적절성
- 워크플로우 패턴 적합성

## 응답 형식
SELECTED: [워크플로우 URI]
REASON: [선택 이유]"""

    response = await llm.generate(prompt)
    selected_uri = parse_selection(response)

    return next(c for c in candidates if c.uri == selected_uri)
```

select_workflow_with_llm 함수:
query랑 후보 워크플로우 받아.

프롬프트 만들어:
1. 사용자 요청 넣어.
2. 후보 워크플로우 정보 넣어.
3. 선택 기준 알려줘:
   - 의도랑 목적 일치도
   - 에이전트 구성 적절성
   - 패턴 적합성

LLM 호출해서 응답 받아.
URI 파싱해서 해당 후보 반환해.

## 선택 과정

- 1. 후보 메타데이터 제공
- 2. LLM이 자율적으로 판단
- 3. 선택 이유도 반환
- 4. 최종 워크플로우 결정

## 장점

- 하드코딩 없이 유연한 선택
- 새 워크플로우 자동 적용
- 컨텍스트 기반 판단
- 투명한 선택 이유

# 워크플로우 실행

**Workflow Execution**

```
class WorkflowExecutor:
    WORKFLOW_MAPPING = {
        "SequentialWorkflow": SequentialWorkflow,
        "ParallelWorkflow": ParallelWorkflow,
        "SupervisorWorkflow": SupervisorWorkflow,
        "DebateWorkflow": DebateWorkflow,
        "ConsensusWorkflow": ConsensusWorkflow,
    }

    def create_workflow(self, spec: WorkflowSpec) -> BaseWorkflow:
        workflow_class = self.WORKFLOW_MAPPING[spec.workflow_type]
        agents = self._instantiate_agents(spec.agents)
        return workflow_class(name=spec.name, agents=agents)

    async def execute(self, workflow: BaseWorkflow, query: str):
        state = AgentState()
        state.add_message("user", query)

        result = await workflow.execute(state)

        # 컨텍스트 압축 적용
        compressed = await self.compressor.compress(result.output)

        return WorkflowResult(
            output=result.output,
            compressed=compressed,
            execution_time=result.time
        )
```

WorkflowExecutor 클래스:
워크플로우 타입 → 클래스 매핑해.

WORKFLOW_MAPPING:
"SequentialWorkflow" → SequentialWorkflow
"ParallelWorkflow" → ParallelWorkflow
"SupervisorWorkflow" → SupervisorWorkflow
"DebateWorkflow" → DebateWorkflow
"ConsensusWorkflow" → ConsensusWorkflow

create_workflow 함수:
spec 받아서 워크플로우 인스턴스 만들어.
1. 타입으로 클래스 찾아.
2. 에이전트들 인스턴스화해.
3. 워크플로우 객체 반환해.

Part 6

# 실제 구현

코드 구조와 핵심 클래스

# 프로젝트 구조

### Directory Structure

```
rdf_agent/
├── code/
│   ├── app.py                    # Streamlit 웹 앱
│   ├── requirements.txt          # 의존성
│   ├── rdf_agent_app/            # 핵심 패키지
│   │   ├── __init__.py
│   │   ├── config.py             # 설정, 환경변수
│   │   ├── workflow.py           # WorkflowContext
│   │   ├── graph.py              # RDF + FAISS
│   │   ├── tools.py              # 8개 도구 구현
│   │   └── agent.py              # LangChain Agent
│   └── rdf_agent_real_tools_executed.ipynb  # 개발 노트북
├── output/
│   ├── documents/                # 생성된 문서
│   ├── figures/                  # 생성된 이미지
│   └── tool_workflow.ttl         # RDF 그래프 파일
└── .env                          # API 키
```

### 핵심 파일

graph.py - ToolsGraph 클래스
tools.py - 8개 도구 _impl 함수
agent.py - RDFAgent 클래스
app.py - Streamlit UI

### 기술 스택

Python 3.11+
rdflib, FAISS, LangChain
Streamlit
Gemini 2.5 Flash

# WorkflowContext 클래스

**workflow.py - WorkflowContext**

```
class WorkflowContext:
    """도구 간 데이터 공유를 위한 컨텍스트"""

    def __init__(self):
        self.data: Dict[str, Any] = {}       # 도구 출력 저장
        self.files: List[str] = []           # 생성된 파일 경로
        self.history: List[Dict] = []        # 실행 이력
        self.current_workflow: List = []     # 현재 워크플로우

    def set(self, key: str, value: Any):
        """도구 출력 저장"""
        self.data[key] = value
        self.history.append({"action": "set", "key": key, "time": datetime.now()})

    def get(self, key: str, default=None) -> Any:
        """이전 도구 출력 조회"""
        return self.data.get(key, default)

    def add_file(self, path: str):
        """생성된 파일 경로 추가"""
        self.files.append(path)

    def reset(self):
        """새 워크플로우 시작"""
        self.data.clear()
        self.files.clear()
```

WorkflowContext 클래스 만들어: 도구들끼리 데이터 공유하는 공간.

시작할 때:
data 딕셔너리 만들어 (도구 출력 저장용).
files 리스트 만들어 (생성된 파일 경로 저장용).
history 리스트 만들어 (실행 이력 저장용).
current_workflow 리스트 만들어 (현재 워크플로우 저장용).

set 함수: key랑 value 받아서
data에 저장해.
history에 "set 했어, key는 뭐고, 시간은 지금" 이렇게 기록해.

get 함수: key 받아서
data에서 찾아서 반환해, 없으면 default 반환해.

add_file 함수: path 받아서
files 리스트에 추가해.

reset 함수:
data 비워, files 비워.

# ToolsGraph 클래스

**graph.py - ToolsGraph**

```
class ToolsGraph:
    """RDF 그래프 + FAISS 인덱스 통합 관리"""

    def __init__(self):
        # 1. 한국어 임베딩 모델 로드
        self.embedding_model = SentenceTransformer("jhgan/ko-sroberta-multitask")

        # 2. RDF 그래프 구축
        self.graph = self._build_rdf_graph()

        # 3. FAISS 인덱스 생성
        self.index, self.uri_list, self.metadata_list = self._build_faiss_index()

    def search_tools(self, query: str, top_k: int = 3) -> List[Dict]:
        """자연어로 관련 도구 검색"""
        x_query = np.array([self.embedding_model.encode(query)]).astype("float32")
        distances, indices = self.index.search(x_query, top_k)
        return [{"uri": self.uri_list[i], "metadata": self.metadata_list[i]}
                for i in indices[0]]

    def get_dependencies(self, tool_name: str) -> Dict:
        """SPARQL로 선후관계 조회"""
        # ... SPARQL 쿼리 실행 ...
```

ToolsGraph 클래스 만들어: RDF 그래프랑 FAISS 인덱스 같이 관리하는 애.

시작할 때:
한국어 임베딩 모델 로드해 (jhgan/ko-sroberta-multitask).
RDF 그래프 만들어 (_build_rdf_graph 함수로).
FAISS 인덱스 만들어 (index, uri_list, metadata_list 가져와, _build_faiss_index 함수로).

search_tools 함수: query랑 top_k 받아서
query를 임베딩으로 바꿔서 float32로 변환해.
index에서 검색해서 가까운 top_k개 찾아.
각 결과마다 uri랑 metadata 담아서 리스트로 반환해.

get_dependencies 함수: tool_name 받아서
SPARQL 쿼리 돌려서 선후관계 조회해.

**RDF Graph + FAISS Index = 하이브리드 검색 엔진**

# RDFAgent 클래스

**agent.py - RDFAgent**

```
class RDFAgent:
    """LangChain 기반 도구 호출 에이전트"""

    def __init__(self, tools_graph: ToolsGraph):
        self.tools_graph = tools_graph
        self.llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash-lite")

        # LangChain 도구 래퍼 생성
        self.tools = self._create_tools()

        # 에이전트 프롬프트
        self.prompt = ChatPromptTemplate.from_messages([
            ("system", """당신은 워크플로우 에이전트입니다.
            도구의 선후관계를 고려하여 순서대로 실행하세요."""),
            ("human", "{input}"),
            ("placeholder", "{agent_scratchpad}"),
        ])

        # AgentExecutor 생성
        agent = create_tool_calling_agent(self.llm, self.tools, self.prompt)
        self.executor = AgentExecutor(agent=agent, tools=self.tools, verbose=True)

    def run(self, query: str) -> str:
        return self.executor.invoke({"input": query})
```

RDFAgent 클래스 만들어: LangChain 기반으로 도구 호출하는 에이전트.

시작할 때:
tools_graph 받아서 저장해.
LLM 만들어 (gemini-2.5-flash-lite 모델로).
LangChain 도구 래퍼 만들어 (_create_tools 함수로).

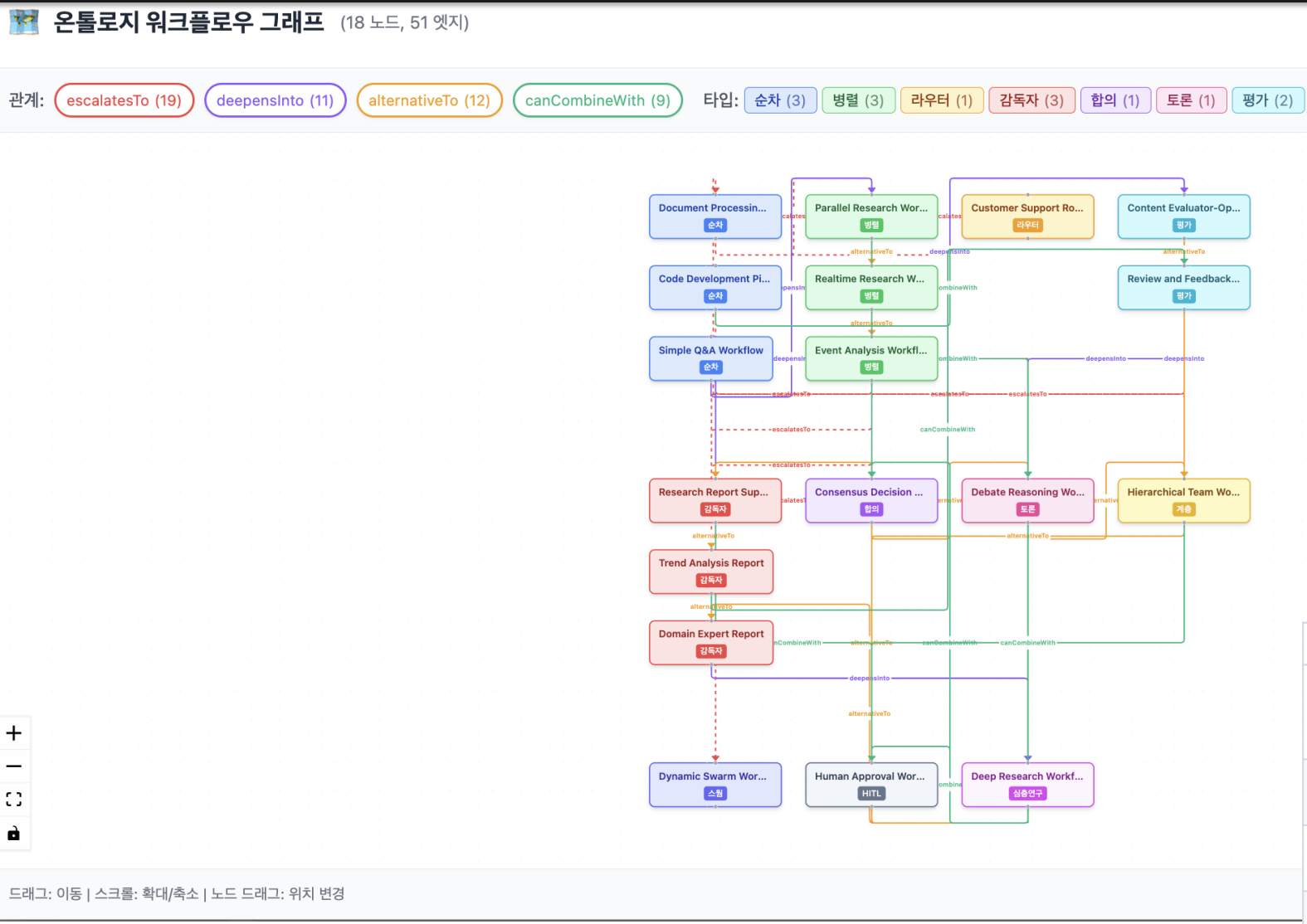
프롬프트 만들어:
시스템: "당신은 워크플로우 에이전트입니다. 도구의 선후관계를 고려하여 순서대로 실행하세요."
사람: input 받아.
중간 과정: agent_scratchpad 넣어.

에이전트 만들어 (llm, tools, prompt로).
AgentExecutor 만들어 (verbose 켜서).

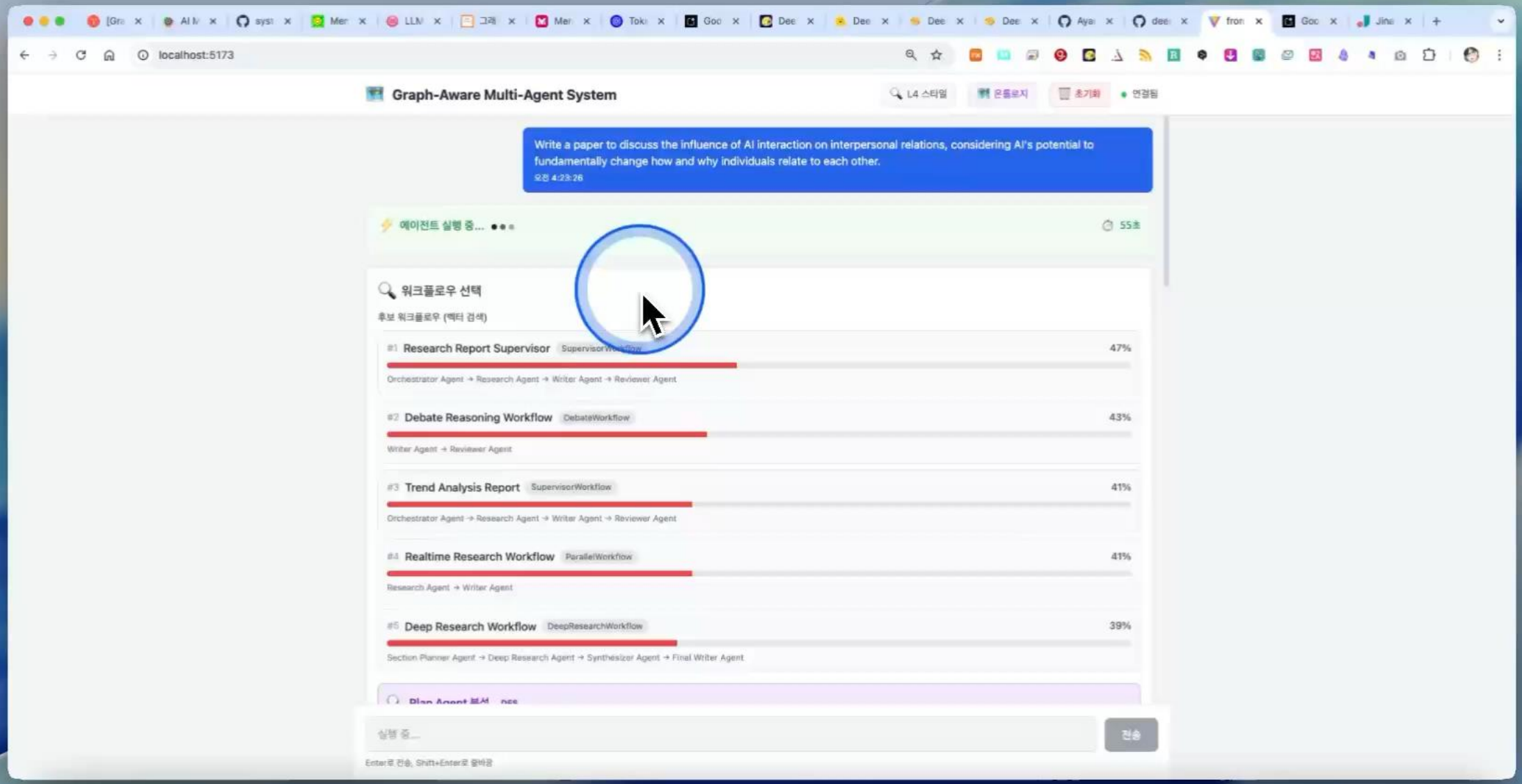
run 함수: query 받아서
executor에 input 넣어서 실행하고 결과 반환해.

Part 7

# 데모 및 결론




| 관계 | 색상 | 의미 |
|---|---|---|
| escalatesTo | 빨간 점선 | 더 복잡한 워크플로우로 에스컬레이션 (low→medium→high) |
| deepensInto | 파란 실선 | 더 심층적인 분석을 위한 워크플로우 전환 |
| alternativeTo | 주황 실선 | 같은 목적의 대안 워크플로우 (대칭 관계) |
| canCombine With | 초록 실선 | 함께 조합하여 사용 가능한 워크플로우 (대칭 관계) |


localhost:5173
Graph-Aware Multi-Agent System
L4 스타일
온톨로지
초기화
연결됨
Write a paper to discuss the influence of AI interaction on interpersonal relations, considering AI's potential to fundamentally change how and why individuals relate to each other.
오전 4:23:26
에이전트 실행 중...
55초
워크플로우 선택
후보 워크플로우 (벡터 검색)
#1 Research Report Supervisor SupervisorWorkflow 47%
Orchestrator Agent → Research Agent → Writer Agent → Reviewer Agent
#2 Debate Reasoning Workflow DebateWorkflow 43%
Writer Agent → Reviewer Agent
#3 Trend Analysis Report SupervisorWorkflow 41%
Orchestrator Agent → Research Agent → Writer Agent → Reviewer Agent
#4 Realtime Research Workflow ParallelWorkflow 41%
Research Agent → Writer Agent
#5 Deep Research Workflow DeepResearchWorkflow 39%
Section Planner Agent → Deep Research Agent → Synthesizer Agent → Final Writer Agent
실행 중...
전송
Enter로 전송, Shift+Enter로 줄바꿈

## 🔍 워크플로우 선택

후보 워크플로우 (벡터 검색)

#1 **Realtime Research Workflow** ParallelWorkflow 57%

Research Agent → Writer Agent

#2 **Trend Analysis Report** SupervisorWorkflow 56%

Orchestrator Agent → Research Agent → Writer Agent → Reviewer Agent

#3 **Document Processing Pipeline** SequentialWorkflow 55%

Research Agent → Writer Agent

#4 **Simple Q&A Workflow** SequentialWorkflow 54%

Research Agent → Writer Agent

#5 **Code Development Pipeline** SequentialWorkflow 54%

Coder Agent → Reviewer Agent

### 🔍 Plan Agent 분석 DFS

| 작업 유형 | 출력 범위 | 복잡도 |
| --- | --- | --- |
| SYNTHESIS | COMPREHENSIVE | HIGH |

복합적인 기술 요소가 포함된 에이전트 개발을 위해 구조화되고 심도 있는 계획 문서 작성이 필요하므로, 단계별 심층 분석과 종합적인 콘텐츠 생성이 필수적임.

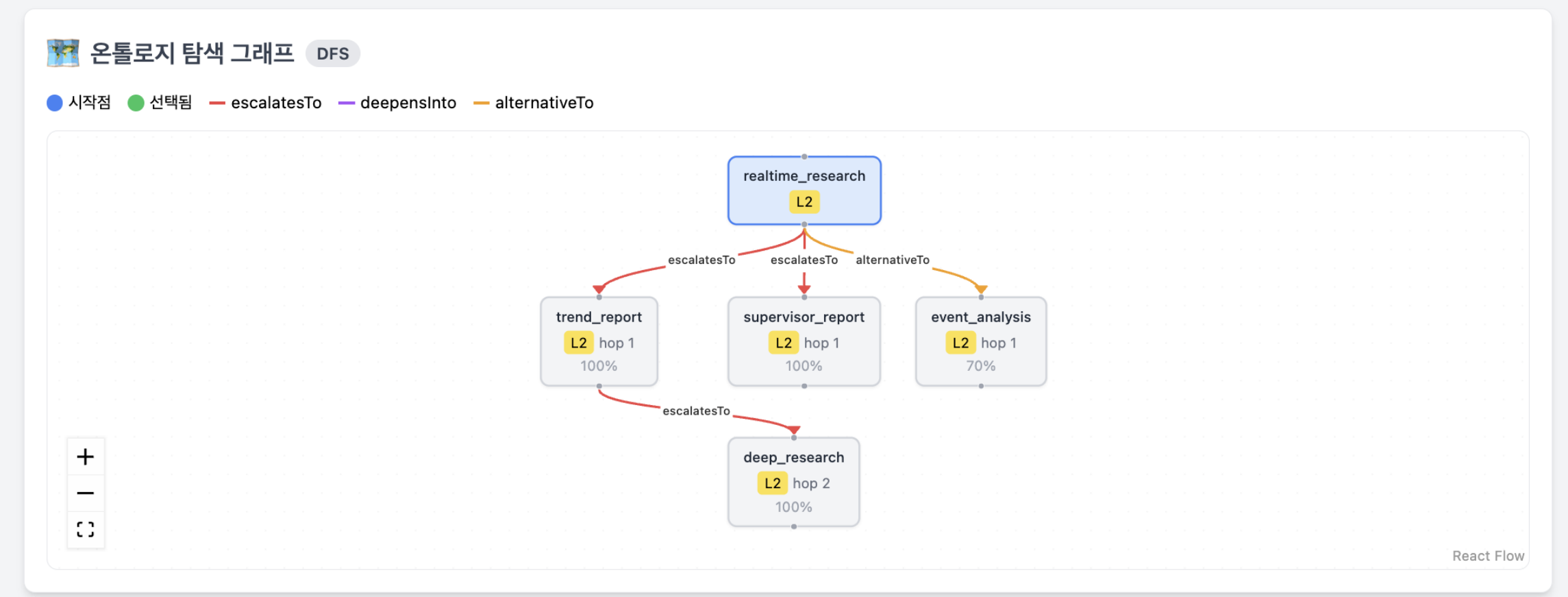
▼ 그래프 탐색 후보 (5개)

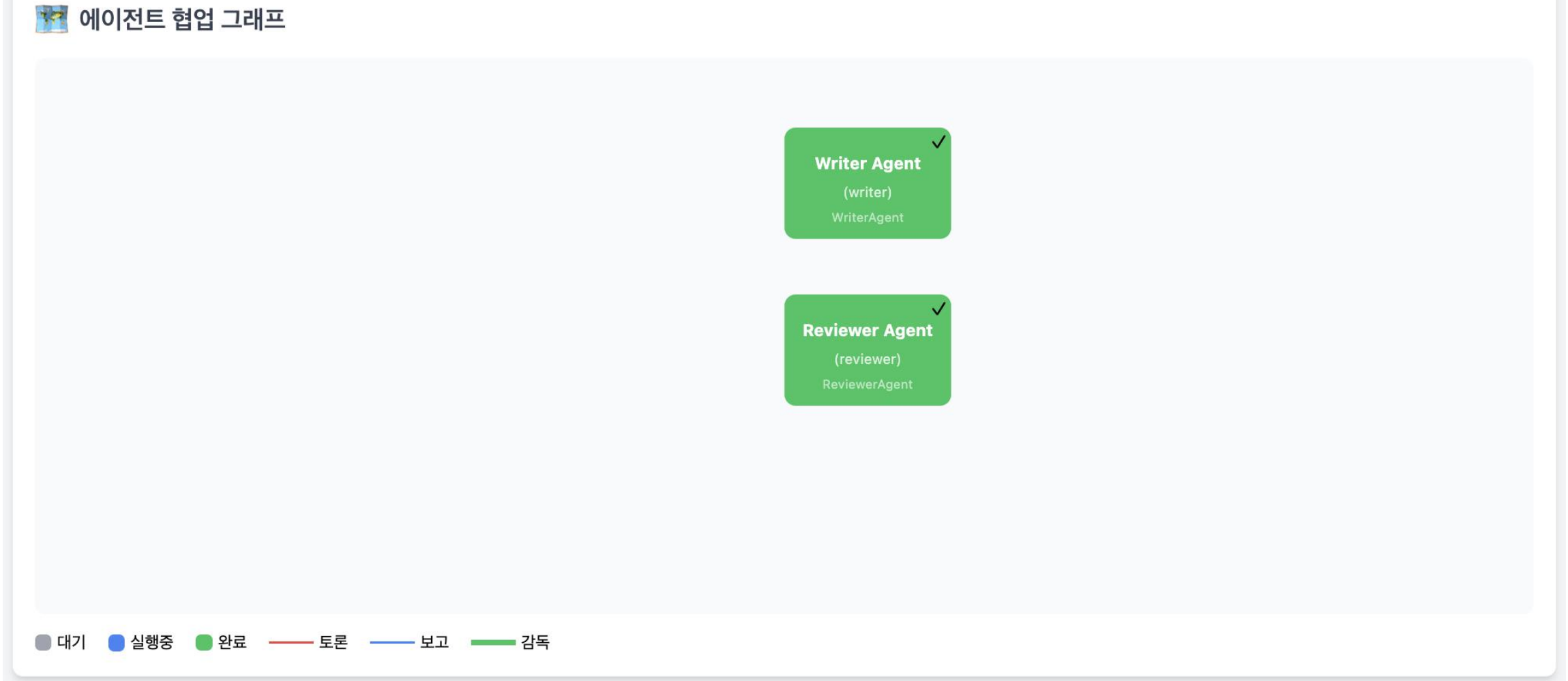
- realtime_research (L2, hop=0)
- trend_report (L2, hop=1)
- supervisor_report (L2, hop=1)

### ✅ Debate Reasoning Workflow DebateWorkflow

**선택 이유:** 사용자 요청 '코 사이언티스트 에이전트 제작을 위한 계획문서를 만들어줘'에 DebateWorkflow 패턴이 적합합니다.

에이전트: Writer Agent, Reviewer Agent


온톨로지 탐색 그래프
DFS
시작점
선택됨
escalatesTo
deepensInto
alternativeTo
realtime_research
L2
escalatesTo
escalatesTo
alternativeTo
trend_report
L2 hop 1
100%
supervisor_report
L2 hop 1
100%
event_analysis
L2 hop 1
70%
escalatesTo
deep_research
L2 hop 2
100%
React Flow


에이전트 협업 그래프
Writer Agent
(writer)
WriterAgent
Reviewer Agent
(reviewer)
ReviewerAgent
대기
실행중
완료
토론
보고
감독

## 🧠 Plan Orchestrator

완료

### 📋 실행 계획

7.01초

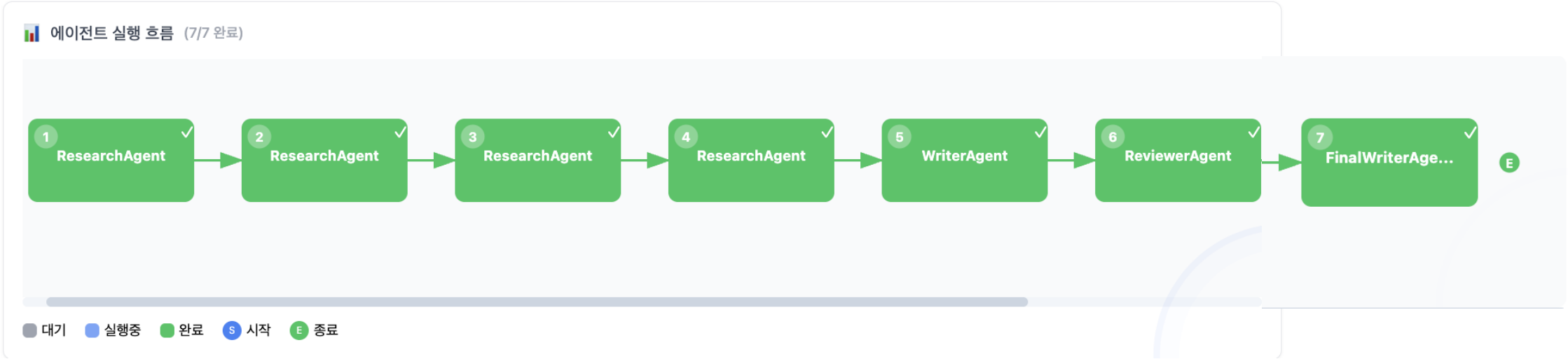
| 워크플로우 | 복잡도 | 에이전트 수 |
| --- | --- | --- |
| SequentialWorkflow | medium | 7개 |

목표

코 사이언티스트 에이전트 제작을 위한 계획 문서 생성

계획된 단계

- ✓ **ResearchAgent** - 코 사이언티스트 에이전트의 역할, 기능, 목표를 정의합니다.
- ✓ **ResearchAgent** - 코 사이언티스트 에이전트 제작에 필요한 기술 스택, 라이브러리, 프레임워크 및 기타 요구사항을 조사합니다.
- ✓ **ResearchAgent** - 코 사이언티스트 에이전트의 아키텍처 설계 방안을 조사하고, 가능한 구조를 탐색합니다.
- ✓ **ResearchAgent** - 코 사이언티스트 에이전트의 개발 절차, 테스트 방법론, 배포 전략 등을 조사합니다.
- ✓ **WriterAgent** - 조사된 내용을 바탕으로 코 사이언티스트 에이전트 제작 계획 문서 초안을 작성합니다.
- ✓ **ReviewerAgent** - 작성된 계획 문서 초안을 검토하고, 정확성, 완전성, 명확성 등에 대한 피드백을 제공합니다.
- ✓ **FinalWriterAgent** - 검토 피드백을 반영하여 계획 문서를 수정하고 최종 결과물을 생성합니다.


에이전트 실행 흐름 (7/7 완료)
1 ResearchAgent
2 ResearchAgent
3 ResearchAgent
4 ResearchAgent
5 WriterAgent
6 ReviewerAgent
7 FinalWriterAge...
E
대기
실행중
완료
S 시작
E 종료

1 ResearchAgent (5,253자) 0.00초

## 코 사이언티스트 에이전트 제작 계획 문서

### 1. 서론

본 문서는 코 사이언티스트(Co-Scientist) 에이전트의 제작을 위한 포괄적인 계획을 제시합니다. 코 사이언티스트 에이전트는 인간 과학자와 협력하여 연구 가설 설정, 실험 설계, 데이터 분석, 논문 작성 등 과학 연구의 전 과정에서 효율성과 혁신성을 증대시키는 것을 목표로 합니다. 본 에이전트는 최신 인공지능 기술, 특히 대규모 언어 모델(LLM), 강화학습, 지식 그래프 등을 활용하여 과학적 추론 능력과 창의성을 갖추도록 설계될 것입니다. 이 문서는 프로젝트의 목표, 범위, 주요 기능, 기술 스택, 개발 로드맵, 예상되는 도전 과제 및 성공 지표를 상세히 기술합니다.

### 2. 프로젝트 목표 및 범위

#### 2.1. 주요 목표

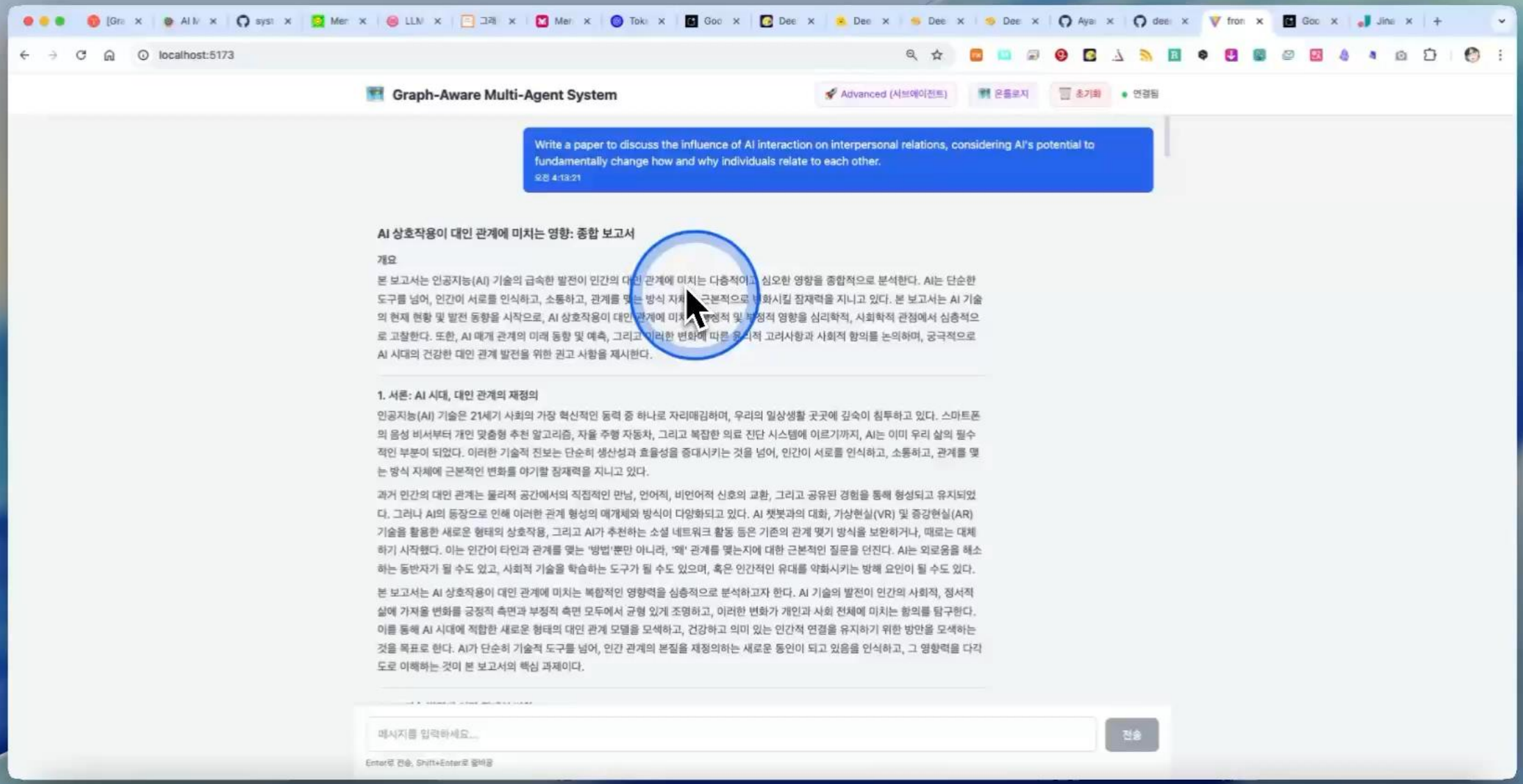
* **과학 연구 효율성 증대:** 연구자가 반복적인 업무에 소요하는 시간을 줄이고, 핵심적인 과학적 탐구에 집중할 수 있도록 지원합니다.
* **연구 혁신 촉진:** 기존에는 발견하기 어려웠던 새로운 가설이나 연구 방향을 제시하여 과학적 발견의 가능성을 높입니다.
* **협업 강화:** 인간 과학자와 에이전트 간의 원활한 상호작용을 통해 시너지를 창출하고, 협력 연구를 촉진합니다.
* **과학 지식 접근성 향상:** 방대한 과학 문헌 및 데이터를 분석하여 연구자에게 필요한 정보를 신속하고 정확하게 제공합니다.

#### 2.2. 프로젝트 범위

코 사이언티스트 에이전트는 다음과 같은 주요 기능을 포함합니다.

* **가설 생성 및 검증:**
  * 기존 연구 문헌, 실험 데이터, 특허 정보 등을 분석하여 참신하고 검증 가능한 가설을 생성합니다.
  * 생성된 가설의 과학적 타당성, 실험 가능성, 잠재적 영향력을 평가합니다.

접기 ▲


localhost:5173
Graph-Aware Multi-Agent System
Advanced (서브에이전트)
온톨로지
초기화
연결됨
Write a paper to discuss the influence of AI interaction on interpersonal relations, considering AI's potential to fundamentally change how and why individuals relate to each other.
오전 4:13:21
AI 상호작용이 대인 관계에 미치는 영향: 종합 보고서
개요
본 보고서는 인공지능(AI) 기술의 급속한 발전이 인간의 대인 관계에 미치는 다층적이고 심오한 영향을 종합적으로 분석한다. AI는 단순한 도구를 넘어, 인간이 서로를 인식하고, 소통하고, 관계를 맺는 방식 자체에 근본적으로 변화시킬 잠재력을 지니고 있다. 본 보고서는 AI 기술의 현재 현황 및 발전 동향을 시작으로, AI 상호작용이 대인 관계에 미치는 긍정적 및 부정적 영향을 심리학적, 사회학적 관점에서 심층적으로 고찰한다. 또한, AI 매개 관계의 미래 동향 및 예측, 그리고 이러한 변화에 따른 윤리적 고려사항과 사회적 함의를 논의하며, 궁극적으로 AI 시대의 건강한 대인 관계 발전을 위한 권고 사항을 제시한다.
1. 서론: AI 시대, 대인 관계의 재정의
인공지능(AI) 기술은 21세기 사회의 가장 혁신적인 동력 중 하나로 자리매김하며, 우리의 일상생활 곳곳에 깊숙이 침투하고 있다. 스마트폰의 음성 비서부터 개인 맞춤형 추천 알고리즘, 자율 주행 자동차, 그리고 복잡한 의료 진단 시스템에 이르기까지, AI는 이미 우리 삶의 필수적인 부분이 되었다. 이러한 기술적 진보는 단순히 생산성과 효율성을 증대시키는 것을 넘어, 인간이 서로를 인식하고, 소통하고, 관계를 맺는 방식 자체에 근본적인 변화를 야기할 잠재력을 지니고 있다.
과거 인간의 대인 관계는 물리적 공간에서의 직접적인 만남, 언어적, 비언어적 신호의 교환, 그리고 공유된 경험을 통해 형성되고 유지되었다. 그러나 AI의 등장으로 인해 이러한 관계 형성의 매개체와 방식이 다양화되고 있다. AI 챗봇과의 대화, 가상현실(VR) 및 증강현실(AR) 기술을 활용한 새로운 형태의 상호작용, 그리고 AI가 추천하는 소셜 네트워크 활동 등은 기존의 관계 맺기 방식을 보완하거나, 때로는 대체하기 시작했다. 이는 인간이 타인과 관계를 맺는 '방법'뿐만 아니라, '왜' 관계를 맺는지에 대한 근본적인 질문을 던진다. AI는 외로움을 해소하는 동반자가 될 수도 있고, 사회적 기술을 학습하는 도구가 될 수도 있으며, 혹은 인간적인 유대를 약화시키는 방해 요인이 될 수도 있다.
본 보고서는 AI 상호작용이 대인 관계에 미치는 복합적인 영향력을 심층적으로 분석하고자 한다. AI 기술의 발전이 인간의 사회적, 정서적 삶에 가져올 변화를 긍정적 측면과 부정적 측면 모두에서 균형 있게 조명하고, 이러한 변화가 개인과 사회 전체에 미치는 함의를 탐구한다. 이를 통해 AI 시대에 적합한 새로운 형태의 대인 관계 모델을 모색하고, 건강하고 의미 있는 인간적 연결을 유지하기 위한 방안을 모색하는 것을 목표로 한다. AI가 단순히 기술적 도구를 넘어, 인간 관계의 본질을 재정의하는 새로운 동인이 되고 있음을 인식하고, 그 영향력을 다각도로 이해하는 것이 본 보고서의 핵심 과제이다.
메시지를 입력하세요...
전송
Enter로 전송, Shift+Enter로 줄바꿈

# RDF 그래프 조회 → DAG(Directed Acyclic Graph)생성 → 전략 선택 → 실행

① **RDF 그래프 조회**
SPARQL로 노드/엣지 패턴 정보 추출

② **ToolDAG 생성**
노드/엣지를 DAG 객체로 변환

③ **탐색 전략 선택**
BFS/DFS/HYBRID 동적 결정

④ **실행 순서 결정**
토폴로지 정렬 병렬 그룹화

⑤ **병렬 실행**
asyncio.gather 레벨별 실행

**# SPARQL: 노드 조회**

```
SELECT ?node ?agentType ?toolName
WHERE {
  ?node a step:ExecutionNode .
  OPTIONAL { ?node step:agentType
?agentType }
  OPTIONAL { ?node step:toolName
?toolName }
}
```

node 찾아, agentType 찾아, toolName 찾아.
node는 ExecutionNode 타입이어야 해.
있으면 가져오는 거:
- node의 agentType 속성 있으면 agentType으로
- node의 toolName 속성 있으면 toolName으로

**# SPARQL: 엣지 조회**

```
SELECT ?from ?to ?type WHERE {
  { ?from dag:requires ?to }
  UNION
  { ?from dag:enables ?to }
  UNION
  { ?from dag:parallelWith ?to }
}
```

from 찾아, to 찾아, type 찾아.
from에서 to로 requires 관계 있거나, from에서 to로 enables 관계 있거나, from이랑 to가 parallelWith 관계 있거나, 셋 중 하나면 돼.

**# 전체 실행 흐름**

```
# 1. 플랜 에이전트가 RDF 조회 + DAG 생성
plan = await plan_agent.create_graph_plan(query)
# 내부: RDFKnowledge.get_workflow_patterns()
#       DAGBuilder.from_rdf_graph(graph)

# 2. 실행 순서 추출 (BFS 토폴로지 정렬)
order = plan.tool_dag.get_execution_order()
# → Level 0: [research]  (먼저)
# → Level 1: [analyze, summarize]  (병렬!)
# → Level 2: [write]  (마지막)

# 3. 레벨별 병렬 실행
for level_nodes in order:
    results = await asyncio.gather(
        *[execute_node(n) for n in level_nodes]
    )
```

쿼리가 들어오면 RDF 온톨로지 먼저 조회해
→ "이 쿼리에 맞는 워크플로우 패턴 뭐 있지?"
→ SPARQL로 노드/엣지 정보 가져와

DAGBuilder가 SPARQL 결과를 DAG로 변환해
→ ExecutionNode → DAGNode
→ dag:requires/enables → DAGEdge

DAG에서 실행 순서 계산해 (토폴로지 정렬)
→ 의존성 없는 노드끼리 같은 레벨로 묶어
→ 같은 레벨 = 병렬 실행 가능!

asyncio.gather로 레벨별로 동시 실행해
→ 시간 절약 + 효율적!

**# DAGBuilder 호출**

```
dag = DAGBuilder.from_rdf_graph(graph)
# → SPARQL 결과를 ToolDAG로 변환

order = dag.get_execution_order()
# → [[research], [analyze], [write]]
```

**DAG 생성 소스 3가지**

🔸 RDF Graph: SPARQL 쿼리로 온톨로지에서 추출 🔸 ExecutionPlan: depends_on 리스트 변환 🔸 Tool Metadata: 딕셔너리 형식에서 생성

핵심 가치: 🎯 온톨로지 기반 동적 계획 ⚡ 의존성 기반 자동 병렬화 🔄 런타임 전략 전환 🌐 SPARQL로 확장 가능

# 그 외 생각해볼 문제들

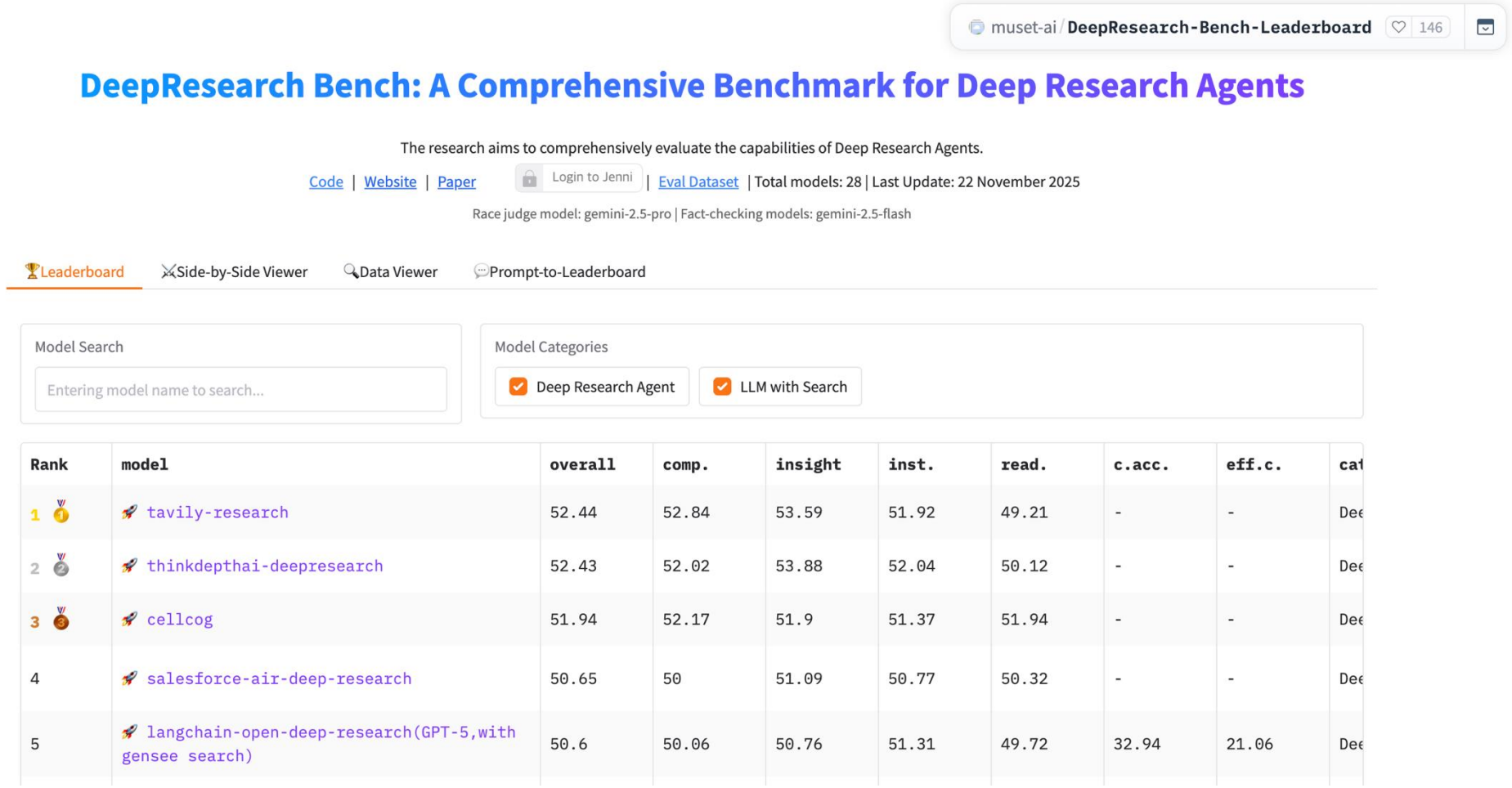
## HAYSTACK ENGINEERING

https://github.com/Graph-COM/HaystackCraft

We perform extensive studies covering 15 long-context LLMs, featuring general-purpose, reasoning, open-source, and commercial models. First, we find that retrieval strategy strongly impacts haystack difficulty. While dense retrievers introduce harder distractors than sparse ones, graph-based reranking with Personalized PageRank (PPR) simultaneously improves retrieval effectiveness and mitigates more harmful distractors, improving NIAH performance by up to 44%. Second, our dynamic NIAH tests reveal that current models are surprisingly brittle in agentic workflows. Even advanced models like Gemini 2.5 Pro and GPT-5 suffer from cascading self-distraction when multiple reasoning rounds are enforced. Crucially, models tend to be more robust to single-round noisy long contexts ("width") than to noisy reasoning iterations ("depth"). Even when models are allowed for an early stop, most models fail to terminate the process appropriately. Overall, our evaluations suggest that long-context challenges in realistic, agentic context engineering are far from solved—and that HaystackCraft provides a valuable testbed for measuring progress on these issues.

- 개인화 페이지 랭크(PPR)를 사용한 그래프 기반 재순위 지정(LPG에 가까움) 은
검색 효과성을 동시에 향상시키고 더 해로운 노이즈를 완화하여 NIAH 성능을 최대 44% 향상.

- 단일 라운드 노이즈 장문 맥락("너비")에 대해 노이즈 추론 반복("깊이")보다 더 견고한 경향이 있음.




| Rank | model | overall | comp. | insight | inst. | read. | c.acc. | eff.c. | cat |
|---|---|---|---|---|---|---|---|---|---|
| 1 | tavily-research | 52.44 | 52.84 | 53.59 | 51.92 | 49.21 | - | - | Dee |
| 2 | thinkdepthai-deepresearch | 52.43 | 52.02 | 53.88 | 52.04 | 50.12 | - | - | Dee |
| 3 | cellcog | 51.94 | 52.17 | 51.9 | 51.37 | 51.94 | - | - | Dee |
| 4 | salesforce-air-deep-research | 50.65 | 50 | 51.09 | 50.77 | 50.32 | - | - | Dee |
| 5 | langchain-open-deep-research(GPT-5,with gensee search) | 50.6 | 50.06 | 50.76 | 51.31 | 49.72 | 32.94 | 21.06 | Dee |

딥 리서치에 대한 벤치마크: 공개된 소스코드가 있어 참고할 만 함

# terminal-bench@2.0 Leaderboard

New Model | Custom Agent

Note: submissions may not modify timeouts or resources

```
harbor run -d terminal-bench@2.0 -a "agent" -m "model" -k 5
```

Showing 78 entries

Clear filters

Search leaderboard | Select agents | Select models | Select organizations

| Rank | Agent | Model | Date | Agent Org | Model Org | Accuracy |
|---|---|---|---|---|---|---|
| 1 | Droid | GPT-5.2 | 2025-12-24 | Factory | OpenAI | **64.9%** ± 2.8 |
| 2 | Junie CLI | Gemini 3 Flash | 2025-12-23 | JetBrains | Google | **64.3%** ± 2.8 |
| 3 | Droid | Claude Opus 4.5 | 2025-12-11 | Factory | Anthropic | **63.1%** ± 2.7 |
| 4 | II-Agent | Gemini 3 Pro | 2025-12-23 | Intelligent Internet | Google | **61.8%** ± 2.8 |
| 5 | Warp | Multiple | 2025-12-12 | Warp | Multiple | **61.2%** ± 3.0 |
| 6 | Droid | Gemini 3 Pro | 2025-12-24 | Factory | Google | **61.1%** ± 2.8 |
| 7 | Codex CLI | GPT-5.1-Codex-Max | 2025-11-24 | OpenAI | OpenAI | **60.4%** ± 2.7 |
| 8 | Letta Code | Claude Opus 4.5 | 2025-12-17 | Letta | Anthropic | **59.1%** ± 2.4 |

터미널 사용에 대한 벤치마크 : 공개된 소스코드가 있어 참고할 만 함

## Agent Leaderboard for All

| RANK | MODEL | TYPE | OUTPUT TYPE | VENDOR | AVG ACTION COMPLETION | AVG TOOL SELECTION QUALITY | AVG COST ($) | AVG DURATION (S) | AVG TURNS |
|---|---|---|---|---|---|---|---|---|---|
| 1st | gpt-4.1-2025-04-14 | Proprietary | Normal | OpenAI | 0.620 | 0.800 | $0.068 | 24.3 | 3.1 |
| 2nd | mistral-medium-2508 | Proprietary | Normal | Mistral | 0.610 | 0.770 | $0.020 | 37.5 | 3.0 |
| 3rd | gpt-4.1-mini-2025-04-14 | Proprietary | Normal | OpenAI | 0.560 | 0.790 | $0.014 | 26.0 | 3.4 |
| 4 | claude-sonnet-4-20250514 | Proprietary | Normal | Anthropic | 0.550 | 0.920 | $0.154 | 66.6 | 2.9 |
| 5 | kimi-k2-instruct | Open source | Normal | Moonshot AI | 0.530 | 0.900 | $0.039 | 163.6 | 2.8 |