생성형 AI 기반 RAG 개발자 과정

Agentic RAG
능동 검색 · 자가교정

Session 29 / 33 — Day 4

검색 결정 · query rewrite · HyDE · 충분성 평가 · self-correction

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면 — 원리

원리

- ✅ **Agentic RAG** 의 5단계 — 결정·재작성·검색·평가·재시도
- ✅ Query rewriting / HyDE / Decomposition 전략
- ✅ **로컬 LLM** 의 도입 가치 — 비용·프라이버시·레이턴시
- ✅ Ollama 아키텍처 — quantization · GGUF · llama.cpp

> 📂 소스: `2.langchain/6.RAG/10~12.*`, `9.rag_web_app/`

이번 시간을 마치면 — 실습

실습

- ✅ LangGraph 로 Agentic RAG 그래프 작성 (`12.agentic_rag.py`)
- ✅ **Query rewrite** + **HyDE** 비교 실험
- ✅ **Ollama** 로컬 모델 설치 + Llama 3 / qwen2.5 / phi3
- ✅ 로컬 RAG — 문서 외부 전송 없이
- ✅ 미니 프로젝트 — Flask + ChromaDB 문서 챗봇

> 📂 소스: `2.langchain/6.RAG/10~12.*`, `9.rag_web_app/`

PART A

PART A
Agentic RAG

> 검색 여부를 결정하는 LLM — 약 25분

두 가지 흐름 비교

고정 RAG (Session 4~6)

```
질문 → 검색 → 답변
```

- 매 질문에 **무조건 검색**
- 검색 결과로 **무조건 답변**
- 단순·고속 — 그러나 한계 존재

한계 사례

- 사용자: "안녕하세요" → 검색 불필요. 인사 응답으로 충분
- 사용자: "Python 의 list 와 tuple 차이" → 검색 결과 부족 시에도 답변 강행 → hallucination
- 사용자: "X 와 Y 의 차이를 비교해줘" → X 와 Y 각각 별도 검색 필요

Agentic RAG

```
질문 → [검색 필요?]
  ├ No  → 즉시 답변
  └ Yes → 쿼리 재작성 → 검색
            → [충분?]
               ├ Yes → 답변
               └ No (iter<2) → 재작성 ↻
```

- LLM 이 **사고 → 행동** 사이클을 스스로 결정
- 검색·재검색·종합·답변 모두 LLM 이 주도
- 추가 호출 비용 vs 정확도 향상

> 🎯 정리: Agentic RAG = **RAG 와 Agent (Day 4 주제) 의 결합**

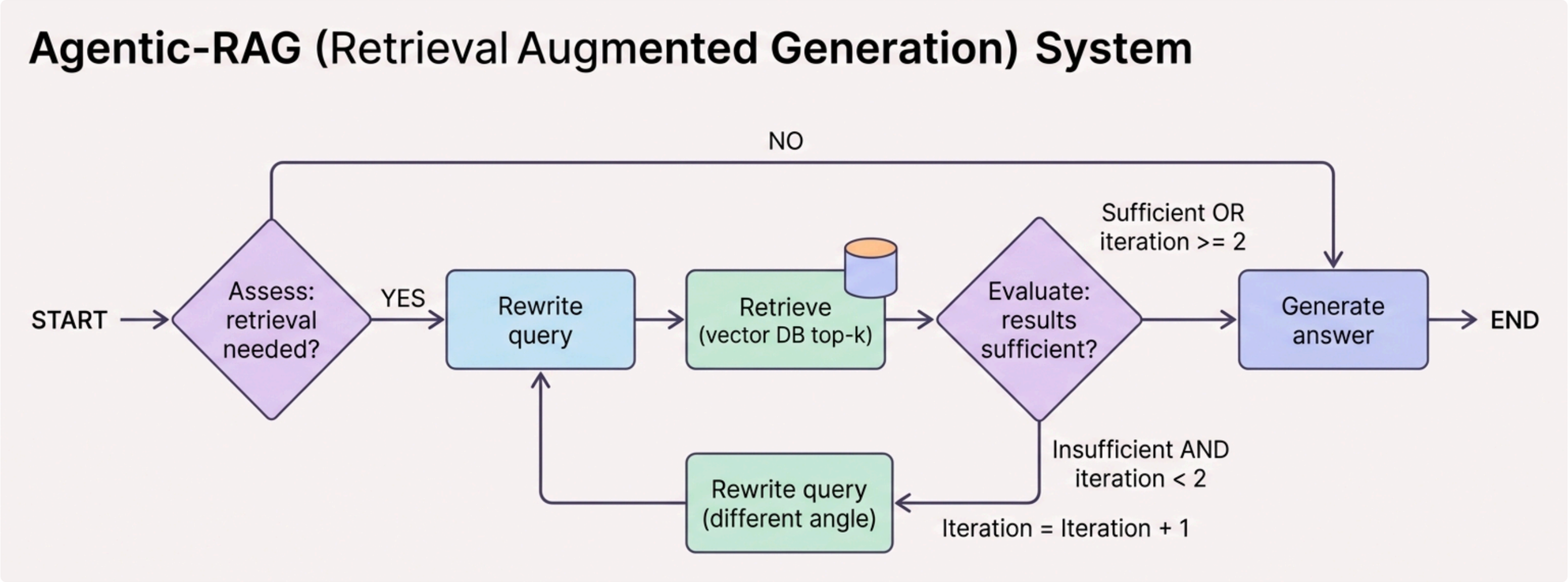
가장 간단한 시작 — 검색을 `@tool` 로

```
@tool
def search_handbook(query: str) → str:
    """사내 규정 검색 (연차/재택/휴가)."""
    hits = vectorstore.similarity_search(query, k=2)
    return "\n".join(d.page_content for d in hits)

agent = create_agent(llm, [search_handbook])
```

- `similarity_search` 를 `@tool` 로 감싸는 것만으로 **Agentic RAG 성립**
- 검색 여부·횟수를 **if 분기가 아닌 LLM** 이 결정
- docstring 의 검색 대상 명시도가 정확도를 좌우

Agentic RAG 루프 전체 구조



Agentic RAG 루프 · 검색 필요? → 쿼리 재작성 → 검색 → 충분? → 부족하면(iter<2) 재작성으로 루프백, 충분하면 응답 생성

12.agentic_rag.py — State + decide·rewrite

```
from typing import TypedDict, List
from langgraph.graph import StateGraph, START, END
from langchain_core.documents import Document

class RAGState(TypedDict):
    question:  str
    rewritten: str   # 재작성된 쿼리
    docs:      List[Document]
    answer:    str
    iteration: int   # 재시도 횟수 (MAX_TRIES=2)

def decide_search(state):
    """검색이 필요한지 LLM 이 판단."""
    decision = llm.invoke(
        f"다음 질문이 외부 문서 검색이 필요한가? yes/no:\n"
        f"질문: {state['question']}"
    ).content.strip().lower()
    return {"needs_search": "yes" in decision}

def rewrite_query(state):
    """검색에 적합하도록 질의 재작성."""
    new_q = llm.invoke(
        f"다음 질문을 벡터 검색에 적합한 키워드 중심으로:\n"
        f"원본: {state['question']}\n"
        f"이전 시도: {state.get('rewritten', '없음')}"
    ).content
    return {"rewritten": new_q}
```

12.agentic_rag.py — retrieve·evaluate·generate

```
def retrieve(state):
    docs = vectorstore.similarity_search(state["rewritten"], k=5)
    return {"docs": docs}

def evaluate_context(state):
    context = "\n".join(d.page_content for d in state["docs"])
    decision = llm.invoke(
        f"다음 컨텍스트로 질문에 답할 수 있는가? yes/no:\n"
        f"질문: {state['question']}\n컨텍스트: {context}"
    ).content.strip().lower()
    return {"sufficient": "yes" in decision}

def generate(state):
    context = "\n".join(d.page_content for d in state["docs"])
    ans = llm.invoke(
        f"질문: {state['question']}\n컨텍스트: {context}"
    ).content
    return {"answer": ans}
```

12.agentic_rag.py — 그래프 구성 + 실행

```
g = StateGraph(RAGState)
g.add_node("decide",    decide_search)
g.add_node("rewrite",   rewrite_query)
g.add_node("retrieve",  retrieve)
g.add_node("evaluate",  evaluate_context)
g.add_node("generate",  generate)

g.add_edge(START, "decide")
g.add_conditional_edges(
    "decide",
    lambda s: "rewrite" if s.get("needs_search")
              else "generate")
g.add_edge("rewrite", "retrieve")
g.add_edge("retrieve", "evaluate")
g.add_conditional_edges(
    "evaluate",
    lambda s: "generate"
              if s.get("sufficient")
              or s["iteration"] ⩾ 2
              else "rewrite")
g.add_edge("generate", END)

app = g.compile()
result = app.invoke({
    "question": "RAG 가 뭐예요?",
    "iteration": 0,
})
print(result["answer"])
```

`12.agentic_rag.py` — 그래프 흐름

그래프 흐름

```
decide → (needs_search?)
  ├─ Yes → rewrite → retrieve → evaluate
  │         └─ (sufficient?)
  │            ├─ Yes → generate
  │            └─ No (iter<2) → rewrite ↻
  └─ No  → generate
```

> 🎯 LangGraph 전체 문법은 **Session 27** 참조. 이번 회차는 구현 가능성 확인에 집중.

사용자 질문 ≠ 좋은 검색 쿼리

전략 1 — Plain Rewrite (가장 단순)

```
원본:    "RAG 가 뭐예요? 그리고 어떻게 만들어요?"
재작성:  "RAG retrieval augmented generation
          구현 방법"
```

```
rewrite_prompt = """검색용 키워드 중심 한 줄로
재작성: {question}"""
```

전략 2 — Decomposition (질문 분해)

```
원본:    "X 와 Y 의 차이를 비교"
분해:    ["X 의 정의", "Y 의 정의",
          "X 와 Y 비교"]   ← 3번 검색
```

```
decompose_prompt = """다음 질문을 검색용
sub-question 3개로 분해:
{question}
한 줄에 하나씩."""
```

전략 3 — HyDE (Hypothetical Document Embeddings)

핵심: 질문 대신 **가상의 답변**으로 검색 → 답변 형태와 더 유사한 chunk 매칭

```
hyde_prompt = """다음 질문에 대한
가상 답변을 3문장으로: {question}"""

hypothetical_answer = llm.invoke(
    hyde_prompt.format(question=q)).content
# hypothetical_answer 로 검색 (질문 자체가 아닌)
docs = vectorstore.similarity_search(
    hypothetical_answer, k=5)
```

효과 (대략적 추정치)

전략	Recall@5 향상	추가 비용
Plain rewrite	+2~5%	LLM 호출 +1회
Decomposition	+5~10% (복합 질문)	LLM +1 + 검색 ×N
HyDE	+5~15% (특수 도메인)	LLM +1회

PART B

PART B
로컬 LLM RAG — Ollama

> 데이터 외부 유출 없는 RAG — 약 20분

4가지 동기

동기	시나리오
프라이버시	의료·법률·기업 내부 — 외부 API 금지
비용	월 $1000 이상이면 로컬 H/W 가 ROI 좋음
레이턴시	LAN 내 호출 ≪ 인터넷 왕복 (50ms vs 500ms)
오프라인	망 분리 환경·해외 출장·격오지

트레이드오프

항목	API	로컬 (Ollama)
품질	gpt-4o = **최상위**	Llama 3 70B = GPT-3.5
속도	빠름 (서버 GPU)	모델 크기 따라
비용	토큰당 과금	**HW 1회** + 전기료
외부 전송	YES	**NO**
모델 선택	제공자 결정	**수백 종 오픈 모델**
한국어	우수	Llama 3+ 양호

본 코스 권장 모델 (2026 기준)

모델	크기	한국어	메모리	적합
llama3.2:3b	3B	보통	8GB	노트북 학습
qwen2.5:7b	7B	**우수**	12GB	**본 코스 권장**
llama3.1:8b	8B	양호	12GB	영어 위주
qwen2.5:32b	32B	매우 우수	32GB+	운영
llama3.3:70b	70B	우수	64GB+	워크스테이션

선택 가이드 한 줄

- **노트북 16GB RAM** → `qwen2.5:7b` (Q4 양자화 = 5GB)
- **워크스테이션 32GB+** → `qwen2.5:32b`
- **GPU 24GB VRAM** → 더 큰 모델도 OK
- **영어 위주** → `llama3.1:8b` 도 좋음

로컬 LLM 설치 및 관리

설치

```
# macOS / Linux
curl -fsSL https://ollama.com/install.sh | sh

# Windows: ollama.com 다운로드

# 모델 받기 (한 번)
ollama pull qwen2.5:7b              # 4.7GB 다운로드

# 실행 확인
ollama run qwen2.5:7b
 >>> Python list comprehension 한 줄 예시
```

모델 관리 명령어

```
ollama list             # 받은 모델 목록
ollama show qwen2.5:7b  # 모델 상세 (크기, 파라미터)
ollama rm qwen2.5:7b    # 모델 삭제
ollama pull llama3.2:3b # 새 모델 받기
ollama serve            # 데몬 실행 (보통 자동)
```

Python 에서 호출

```
# pip install langchain-ollama
from langchain_ollama import ChatOllama
from langchain_core.messages import HumanMessage

llm = ChatOllama(
    model="qwen2.5:7b",
    temperature=0.7,
    num_predict=256,    # max_tokens 대응
    base_url="http://localhost:11434",   # 기본값
)

resp = llm.invoke([HumanMessage("Python 의 list comprehension 한 줄 예시")])
print(resp.content)
```

- LangChain 의 `ChatOllama` — API 모델과 **동일한 인터페이스**
- `base_url` 만 지정하면 원격 Ollama 서버도 동일하게 호출

GGUF · Q4 · Q8 — 표기 체계

원리

- 모델 가중치는 보통 **float16** (각 16 bit)
- 양자화: 16 bit → 8 bit / 4 bit / 2 bit 로 압축
- 정확도 약간 손실 ↔ 메모리·속도 크게 향상

Ollama 의 양자화 표기

태그	의미	크기 (7B 기준)	품질
`qwen2.5:7b` (기본)	Q4_K_M (4-bit)	~4.7GB	표준
`qwen2.5:7b-q8_0`	8-bit	~7.6GB	약간 더 정확
`qwen2.5:7b-fp16`	16-bit (원본)	~14.5GB	원본 품질

GGUF 포맷과 양자화별 추론 속도

GGUF (GPT-Generated Unified Format)

- llama.cpp 가 정의한 모델 파일 포맷
- C++ inference 에 최적화 (CPU·Apple Silicon 빠름)
- HuggingFace 에서 `*.gguf` 파일 직접 받아 Ollama 에 import 가능

```
# 커스텀 모델 import 예
ollama create my-model -f Modelfile
# Modelfile 내용:
#   FROM ./my-model-q4.gguf
#   PARAMETER temperature 0.7
```

양자화별 추론 속도 (M2 Mac, 7B 모델)

정밀도	메모리	tokens/sec
FP16	14GB	~25
Q8	7.6GB	~45
Q4 (기본)	**4.7GB**	**~70**
Q2	2.8GB	~85 (품질 저하 큼)

Ollama + 로컬 임베딩 + Chroma

```
from langchain_ollama import ChatOllama, OllamaEmbeddings
from langchain_chroma import Chroma
from langchain_core.prompts import ChatPromptTemplate
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PyPDFLoader

# ═══ 1. 로컬 임베딩 모델 ===
embeddings = OllamaEmbeddings(model="nomic-embed-text")   # 768d, 한국어 OK
# 또는 ollama pull bge-m3   (1024d, 더 정확)

# ═══ 2. PDF → chunks (외부 전송 없음) ═══
chunks = []
splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=150)
for pdf in Path("./private_docs").glob("*.pdf"):
    chunks.extend(splitter.split_documents(PyPDFLoader(str(pdf)).load()))

# ═══ 3. Chroma 색인 (모두 로컬) ===
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./private_rag",
)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

Ollama + 로컬 임베딩 + Chroma (이어서)

```
# === 4. 로컬 LLM RAG ═══
llm = ChatOllama(model="qwen2.5:7b", temperature=0)

prompt = ChatPromptTemplate.from_messages([
    ("system", "주어진 문서를 근거로만 답하세요. 없으면 '모릅니다'."),
    ("user", "문서:\n{context}\n\n질문: {question}"),
])

def rag(question: str) → str:
    docs = vectorstore.similarity_search(question, k=4)
    context = "\n\n".join(d.page_content for d in docs)
    return (prompt | llm).invoke({"context": context, "question": question}).content

print(rag("내부 보안 정책에서 패스워드 정책은?"))
# → 어떤 데이터도 OpenAI/Anthropic 으로 전송되지 않음
```

검증 — 네트워크 차단으로 테스트

```
# Docker 컨테이너 안에서 외부망 차단
docker run --network=none -v ./private_docs:/docs ollama-rag-image
# 동작하면 = 진짜 로컬 only
```

PART C

PART C
운영 패턴 — 내 도메인에 적용

Flask + Chroma + Ollama/API — 약 15분

학습용 RAG 와 운영 RAG 의 격차

컴포넌트	학습 (Session 4~6)	운영
벡터DB	메모리 / 단일 파일	Chroma server / Qdrant cluster
임베딩	OpenAI API 직접	캐시 + 배치 + 재시도
인덱싱	한 번 실행	**백그라운드 잡** (cron / Celery)
검색	similarity 1개	Hybrid + Rerank
답변	LLM 호출 1번	스트리밍 + citation + caching
평가	수동 검증	RAGAS 자동 + Slack 알림
관측성	print	LangSmith / LangFuse + 사용자 피드백 수집

본 코스 미니 프로젝트 — Flask 풀스택

- `2.langchain/6.RAG/9.rag_web_app/` 가 출발점
- 추가 구성 요소:
- 파일 업로드 → 자동 인덱싱 (`POST /upload`)
- 채팅 히스토리 (`session_id` Session 5 패턴 재사용)
- 스트리밍 응답 (Session 5 SSE 패턴 재사용)
- 출처 패널 (citation chunks UI)

Flask + Chroma — 라우트 + RAG context

```
from flask import Flask, request, Response, jsonify, session
from langchain_chroma import Chroma
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
import uuid, json

app = Flask(__name__); app.secret_key = "dev"
vectorstore = Chroma(persist_directory="./chroma_db",
                     embedding_function=OpenAIEmbeddings())

@app.route("/api/chat/stream", methods=["POST"])
def chat_stream():
    user_q = request.json["question"]
    if "history" not in session:
        session["history"] = []
        session["sid"] = str(uuid.uuid4())

    # 1. RAG — context 준비
    docs = vectorstore.similarity_search(user_q, k=4)
    context = "\n\n".join(
        f"[doc {i+1}] (출처: {d.metadata.get('source')})\n{d.page_content}"
        for i, d in enumerate(docs)
    )

    # 2. 응답 streaming → 다음 슬라이드의 generate()
```

💡 핸들러는 **검색 → context 조립 → 스트리밍 응답** 3단계 구조. Session 5 의 SSE, Session 16 의 citation, Session 14 의 Chroma 가 결합된 형태.

SSE generator — 출처 먼저, 토큰 이어서

```
def generate(user_q, docs, context):
    # 출처 메타 먼저 전송 (UI 출처 패널 즉시 렌더)
    yield f"data: {json.dumps({'type':'sources','sources':[d.metadata for d in docs]})}\n\n"

    # LLM 스트리밍
    llm = ChatOpenAI(model="gpt-4o-mini", temperature=0, streaming=True)
    messages = [
        {"role":"system","content":f"근거 문서로만 답하고 [doc N] 마커. {context}"},
        *session["history"],
        {"role":"user","content":user_q},
    ]
    full = ""
    for chunk in llm.stream(messages):
        if chunk.content:
            full += chunk.content
            yield f"data: {json.dumps({'type':'token','token':chunk.content})}\n\n"

    # 히스토리 누적
    session["history"].append({"role":"user","content":user_q})
    session["history"].append({"role":"assistant","content":full})
    yield "data: [DONE]\n\n"
```

> 💡 **출처를 먼저 yield** 하면 답변 토큰 도착 전에 출처 패널이 채워져 체감 응답성이 향상된다.

PART D

PART D
미니 프로젝트 — 문서 검색 챗봇

> Day 4 통합 — 약 45분

프로젝트 과제 (45분)

목표

담당 분야 문서 (PDF 5개 이상 또는 markdown) 기반 RAG 챗봇 단일 화면 구현.

아키텍처

```
사용자 (브라우저)
    ↓ POST /chat
Flask (Python)
    ├ ChromaDB (./chroma_db)        ← 색인해 둔 컬렉션 그대로
    ├ OpenAI Embedding (질의)
    ├ Cross-Encoder Reranker (옵션)
    └ ChatOpenAI streaming + citation
```

단계

1. **기존 컬렉션 재사용** (5분) — `./chroma_db` 존재 확인
2. **Flask 백엔드** (15분) - `2.langchain/6.RAG/9.rag_web_app/` 베이스 - `/upload` (PDF 업로드 → 자동 인덱싱) - `/chat/stream` (SSE 스트리밍 응답)
3. **프론트엔드** (10분) - 채팅 메시지 영역 - 우측 패널 — 출처 chunk 표시

프로젝트 과제 — Agentic 분기 + 평가

단계 (이어서)

4. **Agentic 분기** (10분, 옵션) - 인사·잡담은 검색 skip - 검색 결과 불충분 시 query rewrite
5. **자체 평가** (5분) - 질문 5개에 대한 답변 + 출처 확인 - faithfulness 점수 자가 평가

산출물

- 동작하는 Flask 앱 (단일 파일 가능)
- 스크린샷 1장 (질문 + 답변 + 출처)
- 평가 메모 — 우수 답변 / 오답 사례 + 추정 원인

핵심 7가지

1. ✅ Agentic RAG = LLM 이 **검색 여부·재작성·평가**를 능동 결정 — `similarity_search` 를 `@tool` 로 감싸는 것이 출발점
2. ✅ 인사·잡담은 **검색 skip**, 필요 시에만 retrieve
3. ✅ Query rewriting 3종 — **Plain · Decomposition · HyDE**
4. ✅ 충분성 평가 → 부족 시 **재작성 후 재검색(iter<2)** — if 분기 → 루프 → LangGraph 3단계
5. ✅ 로컬 LLM(**Ollama**)로도 RAG 구현 가능 — 비용 0·프라이버시
6. ✅ 운영 RAG 6 컴포넌트 (벡터DB·임베딩·인덱싱·검색·답변·평가)
7. ✅ Flask + Chroma + 스트리밍 + citation — 경량 챗봇

실습 과제

필수

- [ ] 문서 챗봇에 **Agentic 분기** 추가 — 인사·잡담은 검색 skip
- [ ] 검색 결과 불충분 시 **query rewrite 후 재검색** 동작 확인
- [ ] 질문 5개에 대한 답변 + 출처 자가 평가 (faithfulness)

옵션

- [ ] `ollama pull qwen2.5:7b` 로 **로컬 LLM RAG** 비교
- [ ] HyDE vs Plain 쿼리 재작성 — 검색 품질 차이 기록