


## SK네트웍스 Family AI 과정 14기

### 모델링 및 평가 LLM 활용 소프트웨어 [1팀]

|        |                                   |
|--------|-----------------------------------|
| 산출물 단계 | 모델링 및 평가                          |
| 평가 산출물 | LLM 활용 소프트웨어                      |
| 제출 일자  | 2025.10.01                        |
| 깃허브 경로 | <a href="#">SKN14-FINAL-1Team</a> |
| 작성 팀원  | 정민영, 안윤지                          |

\*활용한 LLM 모델 파일(구글 드라이브):  [utils](#)

## 1. Api 챗봇(OPENAI LLM) 시스템 개요

본 시스템은 LangGraph를 기반으로 한 대화형 AI 챗봇으로, Google API 문서 관련 질문과 일상 질문을 구분하여 처리하는 구조를 가지고 있습니다.

### [주요 구성 요소]

- LangGraph: 워크플로우 관리 및 노드 기반 처리
- OpenAI GPT 모델:
  - GPT-4o (주요 질의응답, 분류)
  - GPT-4o-mini (일상/불가능 응답)
  - GPT-4.1 (답변 품질 평가)
- Whisper API: OPEN AI의 음성 인식 모델
- ChromaDB: 벡터 데이터베이스 (원문, QA 검색)
- BM25 인덱스: pkl 파일 (원문, QA 검색)
- Hybrid Retriever: ChromaDB + BM25 앙상블
- HuggingFace Embeddings: BAAI/bge-m3 모델

## 2. LangGraph 노드별 상세 구조

### [2.1 analyze\_image 노드]

파일: langgraph\_node2.py

기능:

- 사용자가 업로드한 이미지를 GPT-4o 모델로 분석
- 이미지 내용을 텍스트로 변환하여 후속 처리에 활용

```
def analyze_image(state: ChatState) -> ChatState:
    """ChatState의 이미지를 분석하는 함수"""
    print(f"analyze_image 호출됨 - 이미지 존재: {bool(state.get('image'))}")
    if state.get("image"):
        try:
            # GPT-4 Vision API 호출
            response = client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {
                        "role": "user",
                        "content": [
                            {
                                "type": "text",
                                "text": "이 이미지에 대해 자세히 설명해주세요.",
                            },
                            {
                                "type": "image_url",
                                "image_url": {
                                    "url": state["image"] # URL이면 그대로 사용
                                },
                            },
                        ],
                    },
                ],
                max_tokens=500,
            )

            answer = response.choices[0].message.content
            state["image_analysis"] = (
                answer # 원본 이미지는 유지하고 분석 결과를 별도 필드에 저장
            )
            return state
        except Exception as e:
            print(f"이미지 분석 에러: {str(e)}")
            state["image_analysis"] = f"이미지 분석 중 오류가 발생했습니다: {str(e)}"
            return state
    else:
        return state
```

처리 과정:

1. ChatState의 <image> 필드에서 이미지 데이터 확인
2. 이미지 데이터가 있다면 GPT-4o Vision API 호출하여 이미지 분석결과 생성성
3. 이미지 분석 결과를 ChatState의 <image\_analysis> 필드에 저장

API 설정:

- 모델: gpt-4o
- 최대 토큰: 500

## [2.2 classify 노드]

파일: langgraph\_node2.py

기능:

- 사용자 질문을 3가지 카테고리로 분류 (api / basic / none)
- 이미지 분석 결과가 있으면 질문에 포함하여 분류

분류 기준:

- api: Google API 관련 질문
- basic: 일상 질문
- none: 전문 지식 질문 (API 외)

```
# 분류 노드
def classify(state: ChatState): 1 usage jmy0913
    image_text = state.get("image_analysis")
    question = state["question"]
    chat_history = state.get("messages", [])
    chat_history = chat_history[:4]

    # 이미지 분석 결과가 있으면 질문에 포함시킴
    if state.get("image_analysis"):
        question = (
            f"사용자의 이번 질문:{question}"
            + "\n"
            + f'사용자가 이번에 혹은 이전에 첨부한 이미지에 대한 설명: {state.get("image_analysis")}'
        )

    result = classification_chain.invoke(
        {"question": question, "context": chat_history}
    ).strip()

    state["classify"] = result

    return state
```

처리 과정:

1. 이미지 분석 결과와 질문 결합
2. 최근 4개 대화 히스토리 포함
3. 분류 체인(rag2.py의 classification\_chain) 호출하여 결과 저장

## [2.3 route\_from\_classify 함수]

파일: langgraph\_node2.py

기능:

- 분류 결과에 따라 다음 노드로 라우팅
- 조건부 엣지 처리

라우팅 규칙:

- api → extract\_queries (rag답변을 위한 준비단계) 노드
- basic → simple 노드 (일상 답변)
- none → impossible 노드 (답변 불가능하다고 답변)

```
def route_from_classify(state):  
    route = state.get("classify").strip()  
    # classification_chain이 실제로 뭘 반환하는지에 따라 매핑  
    return route
```

## [2.4 extract\_queries 노드]

파일: langgraph\_node2.py

기능:

- 사용자 질문과 대화 히스토리를 통합
- 이미지 분석 결과 포함하여 통합 질문 생성

```
# (1) 사용자 질문 + 히스토리 통합 + 통합된 질문과 쿼리 추출  
def extract_queries(state: ChatState) -> ChatState:  
    user_text = state["question"]  
    image_text = state.get(  
        "image_analysis"  
    ) # 이미지 설명 (이 부분은 이미 전달된 이미지 설명이어야 함)  
  
    # 히스토리에서 최근 몇 개의 메시지를 가져와서 통합 질문을 생성  
    messages = state.get("messages", [])  
  
    # 최근 4개 메시지만 사용  
    history_tail = messages[-4:] if messages else []  
    context = history_tail.copy()  
  
    # 이미지 설명이 없으면 그냥 넘어가기  
    if image_text:  
        # 이미지 설명이 있을 때만 결합  
        integrated_text = f"질문: {user_text}\n이미지 설명: {image_text}"  
    else:  
        # 이미지 설명이 없으면 질문만 결합  
        integrated_text = user_text  
  
    # 통합된 텍스트를 context에 추가  
    context.append({"role": "user", "content": integrated_text})  
  
    # 통합된 질문을 state["rewritten"]에 저장  
    state["rewritten"] = context  
  
    return state
```

처리 과정:

1. 최근 4개 메시지 히스토리 추출
2. 이미지 분석 결과가 있으면 질문에 결합
3. 통합된 질문(메세지 히스토리 + 질문)을 langgraph state의 <rewritten> 필드에 저장

## [2.5 split\_queries 노드]

파일: langgraph\_node2.py

기능:

- 통합된 질문을 여러 개의 검색 쿼리로 분리(벡터 db 검색용 쿼리 정제)
  - 맥락을 고려한 통합 질문 한글 버전, 영어 버전으로 반환
- JSON 형태로 구조화된 질문 리스트 생성

```
# (2) LLM에게 질문 분리를 시킨다
def split_queries(state: ChatState) -> ChatState: 1 usage  👤 jmy0913
    rewritten = state.get("rewritten")

    response = query_chain.invoke({"rewritten": rewritten})
    state["queries"] = response["questions"] # questions 리스트만 저장

    return state
```

처리 과정:

1. query\_chain(rag2.py에서 가져옴) 호출
2. JSON 형태의 질문 리스트 추출
3. Langgraph state의 queries 필드에 저장

## [2.6 vector\_search\_tool 검색 툴]

파일: langgraph\_node2.py

기능:

- 하이브리드 검색(Chroma + BM25 = 앙상블)을 수행하는 도구
- 원문 문서와 QA 문서를 분리하여 검색 결과 반환
- 기본 top-k는 원문 5, QA 20으로 설정

```
@tool
def vector_search_tool(query: str, api_tags: List[str], text_k:int = 5, qa_k:int = 20):
    """
    태그 기반 원문 하이브리드 검색 (Chroma + BM25, 다중 태그 지원)
    """
    retriever = hybrid_retriever_setting(api_tags, text_k)
    retriever_qa = hybrid_retriever_setting_qa(api_tags, qa_k)

    results_text = retriever.get_relevant_documents(query)
    results_qa = retriever_qa.get_relevant_documents(query)

    print(f"[vector_search_tool] hybrid 검색 완료: '{query}', tags={api_tags}")

    # 각 결과에서 page_content만 추출하여 반환
    return {'text': [result.page_content for result in results_text], 'qa': [result.page_content for result in results_qa]}
```

처리 과정:

1. hybrid\_retriever\_setting(원문)과 hybrid\_retriever\_setting\_qa(QA) 호출
2. query와 api\_tags를 기반으로 벡터 DB 검색 실행
  - Dense 검색(Chroma): api\_tags를 메타데이터 필터로 적용하여 관련 문서 검색
  - BM25 검색: api\_tags별 인덱스를 활용하여 태그 단위로 검색
  - 두 결과를 앙상블(Chroma 0.8 + BM25 0.2)하여 최종 결과 산출
3. 검색된 결과에서 page\_content만 추출하여 반환
  - text: 원문 검색 결과
  - qa: QA 검색 결과

## [2.7 tool\_based\_search\_node 노트]

파일: langgraph\_node2.py

```
def tool_based_search_node(state: ChatState) -> ChatState:
    """LLM이 툴을 사용해서 벡터 DB 검색을 수행하는 노트"""
    queries = state.get("queries", [])
    llm_with_tools = llm.bind_tools([vector_search_tool])
    options_str = "\n".join([f"- {k}: {v}" for k, v in GOOGLE_API_OPTIONS.items()])

    print(f"[tool_based_search_node] 실행 - queries={queries}")

    # LLM에게 명시적으로 "각 질문마다 툴 호출"을 요구
    search_instruction = f"""
    다음의 Google API 관련 **검색 쿼리**들에 대해, 각 쿼리마다 반드시 한 번씩
    `vector_search_tool`을 호출해 주세요.
    - 질문들: {queries}
    - 선택 가능한 Google API 태그(1개 이상):
      {options_str}

    규칙:
    1) 각 질문마다 적절한 api_tags(1개 이상)를 선택하세요.
    2) 선택 가능한 api_tags만 메타 필드로 사용하세요
    3) 질문의 내용과 가장 관련성이 높은 태그를 신중하게 선택하세요.

    예시:
    - 질문: 구글 드라이브에서 파일 권한 수정하는 방법
    - 툴 호출: vector_search_tool(query="구글 드라이브 파일 권한 수정", api_tags=["drive"])

    툴 인자 예:
    {{"query": "<하나의 질문>", "api_tags": ["gmail","calendar"]}}
    """

    response = llm_with_tools.invoke(search_instruction)

    # 툴 호출 결과 추출
    search_results = []
    qa_search_results = []
    tool_calls = []

    if hasattr(response, 'tool_calls') and response.tool_calls:
        for tool_call in response.tool_calls:
            if tool_call['name'] == 'vector_search_tool':
                # 툴 실행
                args = tool_call['args']
                if state['retry']:
                    args['text_k'] = 15
                    args['qa_k'] = 30
                result = vector_search_tool.invoke(args)
                qa_results = result['qa']
                text_results = result['text']
                search_results.extend(text_results)
                qa_search_results.extend(qa_results)
                tool_calls.append({
                    'tool': 'vector search tool',
                    'args': tool_call['args'],
                    'result': result
                })

    if not state['retry']:
        state['search_results'] = list(dict.fromkeys(search_results))
        state['qa_search_results'] = list(dict.fromkeys(qa_search_results))
    else:
        state['hyde_text_results'] = list(dict.fromkeys(search_results))
        state['hyde_qa_results'] = list(dict.fromkeys(qa_search_results))

    state['tool_calls'] = tool_calls

    return state
```

처리 과정:

1. LLM(gpt-4.1)을 vector\_search\_tool과 바인딩
2. 각 query에 대해 반드시 툴 호출하도록 프롬프트 생성
  - queries를 보고 API 태그 중 적합한 것 선택
  - vector\_search\_tool(query, api\_tags=[...], text\_k, qa\_k) 실행
3. 툴 실행 결과에서 search\_results(원문)와 qa\_search\_results(QA) 추출
  - 재실행일 경우(retry=True), 검색 top-k 증가 (text\_k=15, qa\_k=30)
  - 재실행 결과는 hyde\_text\_results, hyde\_qa\_results에 저장

최종 업데이트:

- 검색 결과는 중복 제거 후 state에 저장
  - state['search\_results'], state['qa\_search\_results']
  - 재실행: state['hyde\_text\_results'], state['hyde\_qa\_results']
- state['tool\_calls']에 툴 실행 로그 저장
- 다음 노트(basic\_langgraph\_node)에서 검색 결과를 활용

## [2.8 basic\_langgraph\_node 노트]

파일: langgraph\_node2.py

```
def basic_langgraph_node(state: ChatState) -> Dict[str, Any]:
    search_results_text = state['search_results']
    search_results_qa = state['qa_search_results']
    search_results_text2 = []
    search_results_qa2 = []
    if state['retry']:
        search_results_text2 = state['hyde_text_results']
        search_results_qa2 = state['hyde_qa_results']

    history = state['messages'][-4:]
    question = state['question']

    # 이미지 분석 결과가 있으면 질문에 포함시킴
    if state.get("image_analysis"):
        question = (
            f"사용자의 이번 질문:{question}"
            + "\n"
            + f'사용자가 이전에 혹은 이전에 첨부한 이미지에 대한 설명: {state.get("image_analysis")}'
        )

    # 검색된 결과를 바탕으로 답변 생성
    answer = basic_chain.invoke(
        {
            "question": question,
            "context_text": "\n".join([str(res) for res in search_results_text]),
            "context_qa": "\n".join([str(res) for res in search_results_qa]),
            "context_text2": "\n".join([str(res) for res in search_results_text2]),
            "context_qa2": "\n".join([str(res) for res in search_results_qa2]),
            "history": history,
        }
    ).strip()

    state['search_results_final'] = search_results_text + search_results_qa + search_results_qa2 + search_results_text2
    state['answer'] = answer

    return state # 답변을 반환
```

먼저 langgraph의 state에서 가져오는 값들:

- search\_results\_text: tool\_based\_search\_node에서 얻은 원문 검색 결과
- search\_results\_qa: tool\_based\_search\_node에서 얻은 QA 검색 결과
- hyde\_text\_results: 재검색 시 원문 검색 결과
- hyde\_qa\_results: 재검색 시 QA 검색 결과
- history: 이전 대화 내용을 가져와서 맥락 유지 (최근 대화 4개)
- question: 사용자가 입력한 원본 질문

처리 과정:

1. retry=True인 경우 Hyde 검색 결과를 추가로 병합
2. 이미지 분석 결과(image\_analysis)가 있으면 질문에 결합
  - state에 image\_analysis가 있다면, 해당 내용(이미지 분석 결과)를 질문에 포함
  - 원본 질문과 이미지 설명을 결합하여 더 풍부한 질문 컨텍스트 생성
  - 예: "이 API는 어떻게 사용하나요?" + "이미지: 코드 스크린샷 분석 결과"
3. 모든 검색 결과를 RAG 체인(basic\_chain)에 전달하여 최종 답변 생성



최종 답변 생성 단계:

- **basic\_chain** (RAG 체인)을 호출하여 최종 답변 생성
- 입력 데이터:
  - **question**: 이미지 분석이 포함된 통합 질문
  - **context\_text**: 원문 검색 결과
  - **context\_qa**: QA 검색 결과
  - **context\_text2**: [재실행] Hyde 원문 결과
  - **context\_qa2**: [재실행] Hyde QA 결과
  - **history**: 이전 대화 맥락
- 출력: 검색된 문서를 기반으로 한 전문적인 답변

상태 업데이트 및 반환:

- 최종 답변을 **State**의 **answer** 필드에 저장
- 다음 노드나 최종 응답에서 활용 가능함(**State['answer']**)
- **state['search\_results\_final']**에 전체 검색 결과 저장 : 평가용

## [2.9 evaluate\_answer\_node 노트]

파일: langgraph\_node2.py

```
def evaluate_answer_node(state: ChatState) -> str:
    """
    답변 품질 평가 후, 결과 문자열("good"/"bad")을 반환.
    """
    answer = state["answer"]
    history = state.get("messages", [])
    question = state["question"]

    context = "\n".join(state.get("search_results", [])) # 원본 문서
    context_qa = "\n".join(state.get("qa_search_results", [])) # QA 문서

    result = quality_chain.invoke({
        "history": history[-4:],
        "question": question,
        "context": context,
        "context_qa": context_qa,
        "answer": answer,
    }).strip()

    state["answer_quality"] = result

    if state.get("classify") in ["basic", "none"]:
        state["answer_quality"] = "final"
    elif result == "good":
        state["answer_quality"] = "good"
    elif state.get("retry", False):
        state["answer_quality"] = "final"
    else:
        state["answer_quality"] = "bad"

    print(f"[evaluate_answer_node] 최종 : {state['answer_quality']}")

    return state
```

state에서 가져오는 값들:

- **answer**: **basic\_langgraph\_node**에서 생성된 최종 답변
- **history**: 최근 대화 히스토리 (최대 4개)
- **question**: 사용자의 원본 질문
- **search\_results**: 원문 검색 결과
- **qa\_search\_results**: QA 검색 결과

처리 과정:

1. **answer\_quality\_chain**(평가 체인, **gpt-4.1**)을 호출
  2. 답변의 품질을 평가하여 결과(**good**, **bad**, **final**) 반환
- 기본 평가 기준:
    - 검색 결과를 반영했는지
    - 회피성/무응답 멘트 포함 여부
    - 질문과 맥락 적합성
3. 평가 결과를 **state['answer\_quality']**에 저장

최종 업데이트 및 반환:

- 답변 유형에 따라 분기 처리
  - 평가 결과가 **good**이면 그대로 확정
  - **retry=True** 상태라면 무조건 **final** 처리 : 재실행은 1회만 수행
  - **basic/none** 분류 질문은 항상 **final** 처리 : 분류 오류 대비
  - 그 외에는 **bad**로 표시
- 이후 그래프 흐름:
  - **good/final** → **END**
  - **bad** → **generate\_alternative\_queries** 노드로 이동

## [2.10 generate\_alternative\_queries 노드]

파일: langgraph\_node2.py

```
def generate_alternative_queries(state: ChatState) -> ChatState:
    if state.get("retry", False):
        # 이미 한 번 fallback을 돌았다면 재실행하지 않음
        return state

    question = state['question']
    history = state.get("messages", [])[-4:]

    response = alt_query_chain.invoke({
        "history": history,
        "question": question,
    })

    new_queries = response.get("docs", [])

    print("[generate_alternative_queries] 생성된 쿼리:", new_queries)

    state["queries"] = new_queries

    state["retry"] = True

    return state
```

state에서 가져오는 값들:

- question: 사용자의 원본 질문
- history: 최근 대화 히스토리 (최대 4개)
- retry: 재검색 여부

처리 과정:

1. retry=False일 때만 실행: 이미 한 번 fallback을 돌았으면 실행하지 않음
2. alt\_query\_chain(대체 쿼리 생성 체인, gpt-4o)을 호출
  - 입력 데이터:
    - history: 최근 대화
    - question: 사용자 질문
  - 출력: 한글/영문 가상 답변 2개 (각 2~3문장)
    - 이때 생성되는 것은 검색 재시도를 위한 가상 답변: **Hyde** 방식
3. 생성된 가상 답변들을 state['queries']에 저장하여 이후 검색 입력으로 활용

최종 업데이트 및 반환:

- state['queries']: 새로운 검색용 가상 답변으로 교체
- state['retry'] = True 설정 (재검색 모드 진입)
- 이후 tool\_based\_search\_node로 돌아가 검색 재실행

## [2.11 simple 노드]

파일: langgraph\_node2.py

기능:

- 일상 질문에 대한 친근한 답변 생성
- 전문 지식 질문은 거부

```
# (5) 일상 질문 답변 노드
def simple(state: ChatState):
    print("일상 질문 답변 노드 시작")
    image_text = state.get("image_analysis")
    question = state.get("question")
    chat_history = state.get("messages", [])
    chat_history = chat_history[:4]

    # 이미지 분석 결과가 있으면 질문에 포함시킬
    if state.get("image_analysis"):
        question = (
            f"사용자의 이번 질문:{question}"
            + "\n"
            + f"사용자가 이번에 혹은 이전에 첨부한 이미지에 대한 설명: {state.get('image_analysis')}"
        )

    # 검색된 결과를 바탕으로 답변 생성
    answer = simple_chain.invoke(
        {
            "question": question,
            "context": chat_history,
        }
    ).strip()

    state["answer"] = answer

    return state # 답변을 반환
```

처리 과정:

1. 이미지 분석 결과 포함하여 질문 구성
2. simple\_chain(rag2.py 에서 가져옴)으로 답변 생성
3. 일상 질문이 아닌 경우 "대답할 수 없어요" 응답

## [2.12 impossible 노드]

파일: langgraph\_node2.py

기능:

- 전문 지식 질문에 대한 거부 응답
- Google API 외의 기술 질문 처리

```

# (5) 답변할 수 없는 질문(구글 api 혹은 일상 질문 아닌 경우)
def impossible(state: ChatState):
    print("답변 불가 노드 시작")
    image_text = state.get("image_analysis")
    question = state["question"]
    chat_history = state.get("messages", [])
    chat_history = chat_history[:4]

    # 이미지 분석 결과가 있으면 질문에 포함시킬
    if state.get("image_analysis"):
        question = (
            f"사용자의 이번 질문:{question}"
            + "\n"
            + f"사용자가 이전에 혹은 이전에 첨부한 이미지에 대한 설명: {state.get('image_analysis')}"
        )

    # 검색된 결과를 바탕으로 답변 생성
    answer = imp_chain.invoke(
        {
            "question": question,
            "context": chat_history,
        }
    ).strip()

    state["answer"] = answer

    return state # 답변을 반환

```

처리 과정:

1. 이미지 분석 결과 포함하여 질문 구성
2. `imp_chain(rag2.py에서 가져옴)`으로 거부 메시지 생성
3. "모르는 내용입니다" 형태의 응답

## 3. RAG 체인 구조 (rag2.py)

### [3.1 basic\_chain\_setting]

파일: rag2.py

기능: Google API 문서 기반 전문 답변 생성

```
def basic_chain_setting():
    llm = ChatOpenAI(model="gpt-4o", temperature=0)

    basic_prompt = PromptTemplate.from_template(
        """
당신은 api 문서 관련 전문 챗봇으로서 사용자의 질문에 정확하고 친절하게 답변해야 합니다.
아래 제공되는 문서에 없는 내용은 절대 답변에 포함하지 말고,
원문 문서 내용과 QA 문서 내용 내에서만 답변 내용을 찾아서 제공하세요. 검색 결과를 논리적으로 조합하여 답변을 구성하세요.
원문 추가 문서와 QA 추가 문서가 있다면, 해당 내용까지 참고하여 답변을 제공해주세요.
### 절대 금지 사항
- 문서에 **명시되지 않은 내용**은 추론하거나 일반 지식을 가져와서 설명하지 마세요.
- 문서에 없는 정보를 '추측', '상식', '추론', '일반 규칙'으로 보충하지 마세요.
- 답변에 포함된 모든 사실은 반드시 문서에서 근거를 찾을 수 있어야 합니다.

만약 사용자 질문이 구글 api 문서에 대한 질문이 아니라면, 아래 문서는 무시하고 일상 질문에 대해서만 답변하세요.
일상 질문은 전문적인 내용이 아니고, 코딩과도 관련 없고, 전문지식은 전혀 쓰지 않는 단순한 일상질문입니다.

예를 들어 gitflow가 뭔지 물어보면 답변할 수 없다고 해야 합니다.

원문 문서 : {context_text}
QA 문서 : {context_qa}
원문 추가 문서 : {context_text2}
QA 추가 문서 : {context_qa2}

이전 대화 내역 : {history}
이번 사용자 질문 : {question}

추가로, 이번 사용자 질문에 이미지 분석 내용이 들어있고 사용자가 '이미지에 대한 질문을 하거나' 혹은 '이건 언제?' 와 같이 물어보다면 해당 이미지 분석 결과를 참고해서 답변해주세요.
실제 답변 외에 프롬프트 내용은 답변에 포함시키지 마세요.
"""
    )

    basic_chain = basic_prompt | llm | StrOutputParser()

    return basic_chain
```

프롬프트 특징:

- API 문서 전문 챗봇 역할
- 문서 외 내용 금지
- 이미지 분석 결과 활용
- 대화 히스토리 고려
- 원문/QA/추가 문서 모두 활용 가능

모델 설정:

- 모델: gpt-4o
- Temperature: 0 (일관성 보장)

## [3.2 query\_setting]

파일: rag2.py

기능: 대화 맥락을 고려한 질문 분리

```
def query_setting():
    llm = ChatOpenAI(
        model="gpt-4o",
        temperature=0,
        model_kwargs={"response_format": {"type": "json_object"}},
    )

    query_prompt = PromptTemplate.from_template("""
    유저의 채팅 히스토리와 현재 질문이 주어집니다.

    **중요**: 이전 대화 맥락을 반드시 고려해서 질문을 생성하세요.
    - 현재 질문이 이전 대화와 연관되어 있다면, 이전 맥락을 포함한 통합된 질문을 만들어주세요.
    - 동일한 질문을 한글 질문과 영어 질문 2가지 모두 만들어주세요.

    - 예: 바로 전에 "People API 연락처 조회"에 대해 이야기하고 나서, "그럼 프로필 수정은?"이라는 질문이 나오면 "People API에서 프로필 수정 방법"으로 통합해주세요.
    - 주의사항: 이전에 "People API 연락처 조회"에 대해 이야기하고 나서, "firebase"와 같이 다른 api에 대한 대화 내용이 나온 후 "프로필 수정은?"이라는 질문이 나오면 마지막 대화 맥락에 맞춰서

    - 이전 대화에서 이미 답변이 나온 질문은 생성하지 마세요.
    - 질문은 1개가 될 수도 있고 여러개가 될 수도 있습니다.

    - 만약 사용자 질문에 오타나 잘못된 용어가 포함되어 있다면 올바른 용어로 수정하여 질문을 생성하세요.
    - 예시1 (오타 수정):
      - 사용자 입력: projects.databases.gte 메서드 쓸때 HTTP 리퀘스트 포맷이 뭐예요?
      - 생성 질문: projects.databases.operations.get 메서드의 HTTP 요청 형식은 무엇인가요?
    - 예시2 (오타 수정):
      - 사용자 입력: ttl이 뭐예요?
      - 생성 질문: TTL(Time-to-Live)이 무엇인가요?

    대화 히스토리: {rewritten}

    JSON 반환 형태:
    {{
      "questions": [
        "맥락을 고려한 통합 질문 1 (한글)",
        "맥락을 고려한 통합 질문 2 (한글)",
        "Integrated question considering context 1 (English)",
        "Integrated question considering context 2 (English)"
        ...
      ]
    }}
    """)

    def parse_json(response):
        return json.loads(response.content) # response.content 사용

    chain = query_prompt | llm | parse_json
    return chain
```

프롬프트 특징:

- 이전 대화 맥락 반드시 고려
- 연관 질문 통합 처리
- JSON 형태 응답 강제
- 동일 질문을 한글/영문으로 생성
- 오타 교정 및 올바른 용어로 변환

모델 설정:

- 모델: gpt-4o
- Temperature: 0
- Response Format: JSON Object

## [3.3 classify\_chain\_setting]

파일: rag2.py

기능: 질문 분류 (api/basic/none)

```
def classify_chain_setting():
    llm = ChatOpenAI(model="gpt-4o", temperature=0)

    classification_prompt = PromptTemplate.from_template(
        """
        다음 질문을 분석하여 **주요 목적**에 따라 분류하세요.

        ## 분류 기준
        api: 구글 API 관련 질문이거나, 코딩, 프로그래밍, 보안, 데이터, 네트워킹, IT 기술 등 인터넷 관련 지식에 관한 질문입니다.
            - 질문에 나타나 알 수 없는 단어가 있더라도, **IT 용어처럼 보이고, 질문 맥락상 기술적인 내용을 묻는 경우에만** `api`로 분류하세요.
            - 단순히 이해할 수 없는 글자(예: ㅇㅇㄴㄹ, ㅋㅋㅋ, 아무 의미 없는 문자 나열)는 `api`가 아니라 `none`으로 분류하세요.
        basic: 완전 단순한 일상적인 질문 (날씨, 시간, 간단한 대화)
        none: 일상 질문도 아니고, 구글 API나 코딩, 프로그래밍, 인터넷과 전혀 관계 없는 전문적인 지식에 대한 질문

        ## 출력 규칙
        - 오직 다음 중 하나의 단어만 출력: `api`, `basic`, `none`
        - **절대 추가 설명 없이** 해당 단어만 출력
        - 예: `api`

        최근 대화 내용:
        {context}

        이번 사용자 질문:
        {question}
        """
    )
```

프롬프트에 추가한 예시들

```
## 예시
질문: 구글 캘린더 API에서 이벤트를 어떻게 추가하나요?
정답: api

질문: Google Drive API로 파일 권한을 수정하는 방법 알려줘
정답: api

질문: Gmail API에서 특정 라벨이 붙은 메일만 가져올 수 있어?
정답: api

질문: 구글 맵 API 호출하는 법 알려주고, 참고로 난 지금 배고파
정답: api

질문: 오늘 날씨 어때?
정답: basic

질문: 지금 몇 시야?
정답: basic

질문: 오늘 날씨 어때? 그리고 딥러닝 CNN 구조 설명해줘
정답: none

질문: 구글 캘린더 API 문서 보여줄 수 있어? 아, 그리고 안녕!
정답: api

질문: 양자컴퓨터에서 큐비트 얽힘이 뭔지 설명해줘
정답: none

질문: 딥러닝에서 Transformer 구조가 뭐야?
정답: none
```



질문: **Docker** 컨테이너에서 **MySQL** 볼륨 마운트 하는 방법 알려줘  
정답: **api**

질문: **OpenAI API**랑 **Google API** 차이가 뭐야?  
정답: **api**

질문: **AWS S3 SDK** 사용법 알려줘  
정답: **api**

질문: **Java**에서 문자열을 어떻게 뒤집을 수 있나요?  
정답: **api**

질문: 구글 맵 **API**로 경로 계산하는 방법을 알려줘, 근데 딥러닝을 활용하는 방식으로  
정답: **api**

질문: '**Python**'이라는 언어의 특징을 설명해줘  
정답: **none**

질문: 머신러닝에서 과적합을 방지하는 방법은 뭐야?  
정답: **none**

분류 기준:

- **api**: Google API 질문, 코딩·보안·IT 등 기술 관련 질문 포함
- **basic**: 단순 일상 질문
- **none**: API/일상에 속하지 않는 전문 지식 질문

프롬프트 특징:

- 상세한 예시 제공
- 단일 단어 출력 강제 (**api**, **basic**, **none**)
- 명확한 분류 기준 제시

모델 설정:

- 모델: **gpt-4o**
- Temperature: 0

## [3.4 simple\_chain\_setting]

파일: rag2.py

기능: 일상 질문 전용 답변

```
def simple_chain_setting():
    llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.4)
    simple_prompt = PromptTemplate.from_template(
        """
        너는 사용자의 **일상적인 질문**에만 답변하는 도우미야.

        - 일상적인 질문이면 친절하게 간단히 대답해.
        - 구글 API 질문이나 전문적인 지식 질문이면 ***대답할 수 없어요.*** 라고만 답해야 하는데, 그게 보낸 이미지가 원지 물어보는거나 전에 물어본게 뭐였는지 물어보는거면 대답해줘도 돼.

        최근 대화 내용:
        {context}

        사용자 질문:
        {question}

        ---

        답변:
        """
    )
    simple_chain = simple_prompt | llm | StrOutputParser()
    return simple_chain
```

프롬프트 특징:

- 일상 질문만 처리
- API/전문 질문은 "대답할 수 없어요" 응답
  - 이미지 내용/대화 흐름 관련 질문은 예외적으로 답변 허용
- 친근한 톤 유지

모델 설정:

- 모델: gpt-4o-mini
- Temperature: 0.4 (자연스러운 대화)

## [3.5 impossible\_chain\_setting]

파일: rag2.py

기능: 전문 지식 질문 거부

```
def impossible_chain_setting():
    llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.4)
    imp_prompt = PromptTemplate.from_template(
        """
        너는 사용자의 **사용자의 질문**에 대한 내용을 몰라서 답변할 수 없는 챗봇이야.

        - 최근 대화 내용 및 사용자 질문을 인용해서, **(사용자가 질문한 내용)은 제가 모르는 내용입니다. 일상 질문 혹은 구글 api 관련 질문만 답변드릴수 있어요** 라고만 답해.
        (사용자가 질문한 내용)은 그대로 쓰지 말고 요약하여 잘 정제하여 답변할 때 인용하세요.

        최근 대화 내용:
        {context}

        사용자 질문:
        {question}

        ---

        답변:
        """
    )

    imp_chain = imp_prompt | llm | StrOutputParser()

    return imp_chain
```

프롬프트 특징:

- 질문 내용 인용하여 거부
- 정중한 거부 메시지
- 서비스 범위 명시

모델 설정:

- 모델: gpt-4o-mini
- Temperature: 0.4

## [3.6 answer\_quality\_chain\_setting\_rag]

파일: rag2.py

기능: RAG 답변 품질 평가

```
def answer_quality_chain_setting_rag():
    llm = ChatOpenAI(model="gpt-4.1", temperature=0)

    quality_prompt = PromptTemplate.from_template(
        """
        당신은 RAG 기반 답변 평가자입니다.
        아래는 사용자의 질문과 챗봇의 답변입니다.
        이 답변이 **검색된 결과**, **사용자의 이전 히스토리**, **이전 질문**에 맞게 적절하게 답변했는지 평가하세요.

        평가 기준:
        0. 답변에 '최소하지만', '정보는 제공된 문서에 포함되어 있지 않습니다', '답변할 수 없습니다', '관련된 정보를 찾을 수 없습니다' 와 같이 **부정적, 회피적, 무응답 문구**가 포함되면 무조건
            - 답변이 사용자의 질문에 대해 아무런 구체적 사실을 제공하지 않고 회피성 멘트만 한다면 무조건 "bad"입니다.
            - bad 예시 답변: '최소하지만, GMSMapPoint의 좌표계에서 (0, 0)이 어떤 지점을 의미하는지에 대한 정보는 제공된 문서에 포함되어 있지 않습니다. 다른 질문이 있으시면 말씀해 주세요.'
        1. **검색 결과에 포함되지 않은 정보**를 답변에 포함한 경우, 답변은 "bad"입니다.
        2. **검색 결과와 핵심 내용이 다르게 답변**한 경우, 답변은 "bad"입니다.
        3. **사용자의 히스토리**와 **이전 질문**에 맞지 않는 답변을 한 경우, 답변은 "bad"입니다.
        4. **검색 결과**를 **충실히 반영**하고, **사용자의 과거 질문**이나 **현재 질문**에 맞는 구체적인 답변이 제공되면 "good"입니다.
        5. **검색 결과**를 **무시**하거나 **핵심 내용이 다르게 답변**한 경우, "bad"입니다.

        ---
        사용자의 이전 히스토리: {history}
        사용자의 이전 질문: {question}

        원문 문서 검색 결과: {context}
        QA 문서 검색 결과: {context_qa}
        답변: {answer}
        ---

        출력은 반드시 "good" 또는 "bad" 중 하나만 하세요.
        """
    )

    return quality_prompt | llm | StrOutputParser()
```

프롬프트 특징:

- 답변에 회피성 멘트가 있으면 무조건 bad
- 검색 결과 불일치 시 bad
- 맥락 반영 + 검색 결과 충실 반영 시 good

모델 설정:

- 모델: gpt-4.1
- Temperature: 0

## [3.7 alternative\_queries\_chain\_setting]

파일: rag2.py

기능: [재실행] 가상 답변 생성

```
def alternative_queries_chain_setting():
    llm = ChatOpenAI(
        model="gpt-4o",
        temperature=0,
        model_kwargs={"response_format": {"type": "json_object"}}
    )

    alt_prompt = PromptTemplate.from_template(
        """
        당신은 구글 api 문서에 대한 전문가이고, 당신이 아는 api 주제는 아래 11가지입니다.

        "map": "Google Maps API (구글 맵 API)",
        "firestore": "Google Firestore API (구글 파이어스토어 API)",
        "drive": "Google Drive API (구글 드라이브 API)",
        "firebase_authentication": "Google Firebase API (구글 파이어베이스 API)",
        "gmail": "Gmail API (구글 메일 API)",
        "google_identity": "Google Identity API (구글 인증 API)",
        "calendar": "Google Calendar API (구글 캘린더 API)",
        "bigquery": "Google BigQuery API (구글 빅쿼리 API)",
        "sheets": "Google Sheets API (구글 시트 API)",
        "people": "Google People API (구글 피플 API)",
        "youtube": "YouTube API (구글 유튜브 API)"

        사용자의 이전 대화 내역과 이번 질문의 맥락을 고려하여,
        당신이 아는 내용을 최대한 활용해서 각 답변 당 2~3문장 내로 구체적으로 자세히 답변해주세요.
        다른 방안을 제시하지 말고, 반드시 사용자의 질문 요구를 충족하는 답변을 생성하세요.

        답변은 동일한 내용을 영어와 한글 각각 1개씩 만들어주세요.

        사용자의 이전 히스토리:
        {history}

        사용자의 이번 질문:
        {question}

        ---
        JSON 반환 형태:
        {{
            "docs": [
                "한글 답변",
                "english answer"
            ]
        }}
        """
    )

    def parse_json(response):
        try:
            return json.loads(response.content)
        except Exception:
            # LLM이 JSON 형식을 지키지 못했을 때 빈 리스트 반환
            return {"questions": []}

    chain = alt_prompt | llm | parse_json
    return chain
```

프롬프트 특징:

- 사용자의 질문 요구를 충족하는 한글/영문 짧은 답변 생성 (2~3문장)
- 다른 방안을 제시하지 않고 구체적·자세한 답변 제공
- 생성된 가상 답변은 **Hyde** 방식으로 검색 재시도에 활용

모델 설정:

- 모델: gpt-4o
- Temperature: 0
- Response Format: JSON Object

## 4. 벡터 데이터베이스 및 검색 구조

### [4.1 vector\_db.py / vector\_db\_qa.py]

파일: vector\_db.py, vector\_db\_qa.py

기능: Google API 문서 DB 생성 및 로드 (원문 / QA)

구현 특징:

- 구글 드라이브에서 해당 DB(chroma\_db / qa\_chroma\_db) 다운로드
- 로컬에 chroma.sqlite3 포함 DB 저장
- DB가 존재하지 않으면 자동으로 다시 생성
- 원문 DB는 Google API 공식 문서, QA DB는 질문-답변 전용

### [4.2 retriever.py / retriever\_qa.py]

파일: retriever.py, retriever\_qa.py

기능: Chroma 기반 Dense 검색 (원문 / QA)

구현 특징:

- retriever\_setting(원문), retriever\_setting2(QA) 함수 제공
- DB 디렉토리 상태 확인 후, 없으면 vector\_db.py / vector\_db\_qa.py의 create\_chroma\_db 실행
- DB가 있으면 그대로 Chroma 벡터스토어 로드
- 컬렉션: google\_api\_docs(원문), qna\_collection(QA)
- 임베딩: BAAI/bge-m3
- 로컬 디렉토리 검사 후 DB 로드, 없을 경우 새로 생성

### [4.3 retriever\_bm25.py]

파일: retriever\_bm25.py

기능: 키워드 검색 (BM25)

구현 특징:

- 원문/QA 각각 BM25 인덱스 관리 (bm25\_index.pkl, bm25\_qa\_index.pkl)
- 실행 시 동작:
  - 1. pkl 파일이 존재하면 해당 인덱스를 로드
  - 2. 없으면 retriever\_setting / retriever\_setting2에서 문서·메타데이터를 불러옴
  - 3. 문서를 tags 기준으로 그룹핑

- 4. 각 태그별로 BM25Retriever.from\_documents 실행
- 5. {tag: BM25Retriever} 딕셔너리를 생성하여 pkl로 저장

pkl 내부 구조:

```
{  
  "drive": <BM25Retriever 객체>,  
  "gmail": <BM25Retriever 객체>,  
  "calendar": <BM25Retriever 객체>,  
  ...  
}
```

retriever 접근 함수 제공:

- bm25\_retrievers\_by\_tag (원문, 기본 k=5)
- bm25\_retrievers\_by\_tag\_qa (QA, 기본 k=20)

## [4.4 retriever\_hybrid.py]

파일: retriever\_hybrid.py

기능: Dense + BM25 하이브리드 검색

구현 특징:

- hybrid\_retriever\_setting(원문), hybrid\_retriever\_setting\_qa(QA) 함수
- api\_tags 기반 메타 필터링 지원
- 원문 및 QA 각각 별도 retriever 제공
- Dense(Chroma)와 BM25를 앙상블하여 최종 검색 결과 생성
  - 가중치 기반 앙상블: Chroma 0.8 + BM25 0.2
- 태그가 여러 개일 경우 BM25 retriever를 Ensemble로 결합

검색 방식 (Hybrid Search):

- Dense + BM25 (앙상블)
- Dense : 의미 기반 검색 + api\_tags 메타 필터링
- BM25 : 키워드 기반 검색 + 태그별 인덱스 활용

## 5. 음성 인식 구조 (whisper.py)

### [5.1 call\_whisper\_api 함수]

파일: whisper.py

기능:

- 오디오 파일을 텍스트로 변환
- OpenAI Whisper API 활용

```
def call_whisper_api(audio_file): 2 usages  jmy0913
    try:
        # 임시 파일로 저장
        temp_path = f"temp_audio_{audio_file.name}"
        with open(temp_path, "wb") as f:
            for chunk in audio_file.chunks():
                f.write(chunk)

        # OpenAI 1.0.0+ 방식으로 Whisper API 호출
        client = openai.OpenAI()
        with open(temp_path, "rb") as audio_file_obj:
            transcript = client.audio.transcriptions.create(
                model="whisper-1", file=audio_file_obj, language="ko"
            )

        # 임시 파일 삭제
        os.remove(temp_path)

        return transcript.text

    except Exception as e:
        print(f"Whisper API 에러: {str(e)}")
        return f"음성 인식 중 오류가 발생했습니다: {str(e)}"
```

처리 과정:

1. django 프론트에서 음성을 받으면 blub form객체로 받아서 call\_whiper로 넘겨줌
2. call \_whisper에서 아래와 같이 처리하여 django 백엔드에 반환해줌
  - 임시 파일로 오디오 저장
  - Whisper API 호출
  - 한국어 설정으로 변환
  - 임시 파일 삭제
  - 텍스트 결과 반환
3. Django 백엔드에서는 해당 텍스트로 메인 langgraph로 전달해줌

API 설정:

- 모델: whisper-1
- 언어: ko (한국어)
- 오류 처리 포함



## 6. 메인 실행 구조 (main3.py)

### [6.1 run\_langgraph 함수]

파일: main3.py

기능:

- LangGraph 실행 진입점
- 세션 관리 및 상태 처리

```
def run_langgraph(user_input, config_id, image, chat_history=None):
    try:
        config = {"configurable": {"thread_id": config_id}}

        # chat_history가 None이면 빈 리스트로 초기화
        if chat_history is None:
            chat_history = []

        print(f"run_langgraph 호출 - 입력: {user_input}, 이미지: {bool(image)}")

        result = graph.invoke(
            {
                "messages": chat_history,
                "question": user_input,
                "image": image,
                "retry": False
            },
            config=config,
        )

        # print(f"그래프 실행 결과: {result}")
        return result["answer"]
    except Exception as e:
        print(f"run_langgraph 에러: {str(e)}")
        import traceback

        traceback.print_exc()
        return f"처리 중 오류가 발생했습니다: {str(e)}"
```

처리 과정:

1. 스레드 ID 기반 설정 구성

2. 대화 히스토리 초기화

3. 그래프 실행

- 사용자 입력(`user_input`)을 `state`에 포함하여 `LangGraph`로 전달
- 이미지 있는 경우, 사용자 입력(`user_input`)과 이미지(`image_url`)를 함께 전달
  - `analyze_image` 노드에서 `GPT-4o` 모델을 통해 이미지 분석을 수행하고, 그 결과를 질문 맥락에 결합

4. 답변 추출 및 반환

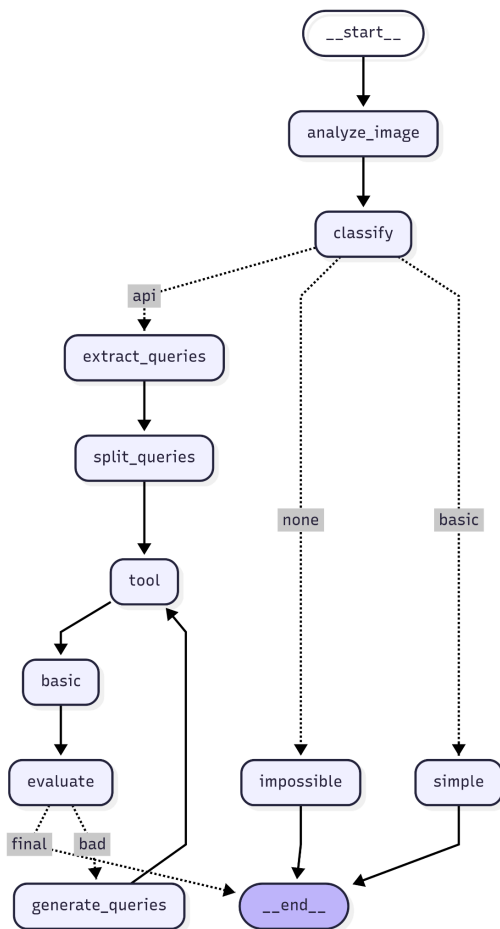
## 7. 그래프 설정 (langgraph\_setting2.py)

### [7.1 graph\_setting 함수]

파일: langgraph\_setting2.py

기능:

- LangGraph 워크플로우 정의
- 노드 연결 및 조건부 라우팅



흐름 구조:

- 시작: analyze\_image → classify
- classify 결과에 따라 분기:
  - api → extract\_queries → split\_queries → tool → basic → evaluate → good/final → END
  - bad → generate\_queries → tool → basic → evaluate → END
  - basic → simple → END
  - none → impossible → END

## 8. 데이터 흐름

### [8.1 ChatState 구조]: Langgraph에서 사용하는 State

```
class ChatState(TypedDict, total=False):
    question: str # 유저 질문
    answer: str # 모델 답변
    rewritten: str # 통합된 질문
    queries: List[str] # 쿼리(질문들)
    search_results: List[str] # 벡터 DB 검색 결과들
    qa_search_results: List[str] # qa 벡터 db 검색 결과들
    messages: List[Dict[str, str]] # 사용자 및 모델의 대화 히스토리
    image: str # 원본 이미지 데이터
    image_analysis: str # 이미지 분석 결과
    classify: str # 질문 분류
    tool_calls: List[Dict[str, Any]] # 도구 호출 기록
    qa_tool_calls: List[Dict[str, Any]]
    answer_quality: str
    retry: bool
    hyde_qa_results: List[str]
    hyde_text_results: List[str]
    search_results_final: List[str]
```

### [8.2 처리 흐름]

1. 입력: 사용자 질문(음성일시에도 처리가능) + 이미지(선택) + 대화 히스토리
2. 이미지 분석: GPT-4o로 이미지 내용 추출
3. 분류: 질문 유형 판별 (api/basic/none)
4. 라우팅: 분류 결과에 따른 노드 분기
5. 검색: Dense(의미 기반, api\_tags 메타 필터링) + BM25(키워드 기반, 태그 인덱스) 앙상블
6. 답변 생성: 적절한 체인으로 최종 답변 생성
7. 답변 품질 평가: good / bad / final으로 답변 평가하여 재실행 여부 결정
8. 재실행: 한글/영문 가상 답변 생성 후, 재검색 결과를 활용해 답변 보강
9. 출력: 사용자에게 답변 전달