📦 개발된 소프트웨어: RAG 기반 LLM과 벡터 데이터베이스 연동 구현 코드

4. RAG 파이프라인 구현

4.1 QueryRouter (질문 검증)

파일: `src/advanced_rag.py` (Line 52-100)

목적: 사용자 질문의 유효성 검증 및 카테고리 분류

코드:

```
class QueryRouter:
    """사용자 쿼리를 검증하고 정제하는 라우터"""

    def __init__(self, llm: ChatOpenAI):
        self.llm = llm
        self.router_prompt = ChatPromptTemplate.from_messages([
            ("system", """
당신은 사용자 질문을 분석하고 정제하는 라우터입니다.

작업:
1. 질문이 의미 있는지 검증 (인사말, 욕설, 무의미한 입력 제외)
2. 질문 카테고리 분류 (정책검색, 추천, 일반질문 등)
3. LLM이 처리하기 좋은 형태로 정제
4. 만약 질문에 '전국', '전체', '모든', '모두' 등 전국 단위 키워드가 포함되어 있고, 지역명이 명확하지 않으면 refined_query에서 '전국', '전체' 등 지역 관련 키워드를 제거하고 핵심 정책 키워드만 남겨서 더 일반화된 형태로 정제하라. 예를 들어 '전국 일자리' → '일자리 정책', '전국 청년 복지' → '청년 복지 정책' 등으로 정제.
5. refined_query는 반드시 검색에 최적화된 형태로 반환하라.

응답 형식 (JSON):
{{
    "is_valid": true/false,
    "category": "정책검색|정책추천|일반질문|기타",
    "refined_query": "정제된 질문",
    "reason": "판단 이유"
}}

예시:
- 입력: "전국 일자리" → refined_query: "일자리 정책"
- 입력: "전국 청년 복지" → refined_query: "청년 복지 정책"
- 입력: "서울 월세 지원" → refined_query: "서울 월세 지원 정책"
- 입력: "청년 정책" → refined_query: "청년 정책"
"""),
            ("user", "{query}")
        ])

    def route(self, query: str) -> Dict:
```

```
        """쿼리를 검증하고 정제"""
        try:
            response = self.router_prompt | self.llm | StrOutputParser()
            result_str = response.invoke({"query": query})

            # JSON 파싱
            result = json.loads(result_str)

            return result
        except Exception as e:
            return {
                "is_valid": True,
                "category": "일반질문",
                "refined_query": query,
                "reason": "파싱 실패로 원본 사용"
            }
```

특징:

- LLM 기반 질문 검증 및 정제: 사용자 입력의 유효성 검사와 검색 최적화된 `refined_query` 생성(전국/지역 키워드 처리 포함)
- 구조화된 JSON 출력: `is_valid`, `category`, `refined_query`, `reason` 필드로 재현 가능한 응답 제공
- 안전한 폴백: 파싱/LLM 호출 실패 시 원본 쿼리를 사용해 검색 진행
- 내부 구현: `ChatPromptTemplate` + `StrOutputParser` 활용으로 일관된 파싱과 디버깅 가능

4.2 MultiQueryGenerator (다중 쿼리 생성)

파일: `src/advanced_rag.py` (Line 105-160)

목적: 하나의 질문을 3개의 다양한 관점 쿼리로 확장

코드:

```
class MultiQueryGenerator:
    """하나의 질문을 여러 관점의 쿼리로 확장"""

    def __init__(self, llm: ChatOpenAI):
        self.llm = llm

        self.multi_query_prompt = ChatPromptTemplate.from_messages([
            "system", """
당신은 사용자의 원본 질문을 **의도와 핵심 키워드를 유지**한 채 검색에 최적화된 여러 관점의 쿼리로 확장하는 전문가입니다.

**원본 질문의 내용이나 조건을 임의로 추가하거나 변경하지 마세요. 오직 검색 관점만 다양화해야 합니다.**

주어진 질문을 3가지 다른 관점의 검색 쿼리로 재구성하세요:

1. **지역(Region) 추출 강제: 사용자가 지역을 언급하면, '해당 지역 + 전국' 정책만 반환
```

```
합니다.
2.  **정책 키워드(Policy Keyword): 질문의 **핵심 의도**와 관련된 정책 키워드를 추출하
여 관련된 정책만 반환할 것.
3.  **유사한 의미 또는 관련 정책명**을 포함하는 쿼리 (유의어 활용)

각 쿼리는 한 줄로 작성하고, 번호 없이 줄바꿈(\n)으로 구분하세요. """,
            ("user", "{query}")
        ])

    def generate(self, query: str) -> List[str]:
        """다중 쿼리 생성"""
        try:
            response = self.multi_query_prompt | self.llm | StrOutputParser()
            result = response.invoke({"query": query})

            # 쿼리 분리 (줄바꿈 기준)
            queries = [q.strip() for q in result.split('\n') if q.strip()]
            # 원본 쿼리 포함
            all_queries = [query] + queries


            return all_queries

        except Exception as e:
            return [query]
```

특징:

- 검색 관점 확장: 원본 쿼리와 LLM이 생성한 최대 3개의 보조 쿼리(지역/정책/유의어 관점)로 탐색 범위를 넓힘
- 지역 및 정책 키워드 보존 규칙 포함: 사용자가 지역을 명시하면 해당 지역 + 전국(전국범위) 관련 문서를 함께 고려하도록 설계
- 안전한 폴백: LLM 실패 시 원본 쿼리만 사용

4.3 EnsembleRetriever (하이브리드 검색)

파일: `src/advanced_rag.py` (Line 165-310)

목적: BM25 키워드 검색 + 벡터 유사도 검색 결합

코드:

```
class EnsembleRetriever:
    """Dense, BM25 검색을 결합한 앙상블 리트리버"""

    def __init__(
        self,
        documents: List[any],
        vectorstore: Chroma,
        llm: ChatOpenAI = None,
```

```
        bm25_k: int = 5,
        vector_k: int = 10,
        bm25_weight: float = 0.4,
        vector_weight: float = 0.6
    ):
        self.documents = documents
        self.vectorstore = vectorstore
        self.llm = llm

        # 파라미터 저장
        self.bm25_k = bm25_k
        self.vector_k = vector_k
        self.bm25_weight = bm25_weight
        self.vector_weight = vector_weight

        # 각 리트리버 초기화
        self._build_bm25()
        self._build_vector()

    def _build_bm25(self):
        """BM25 Retriever 생성"""
        if not RETRIEVERS_AVAILABLE or BM25Retriever is None:
            self.bm25_retriever = None
            return

        if not self.documents:
            self.bm25_retriever = None
            return

        try:
            # BM25Retriever 초기화 (from_documents 사용)
            self.bm25_retriever = BM25Retriever.from_documents(
                documents=self.documents,
                k=self.bm25_k
            )
        except TypeError as e:
            # from_documents가 실패하면 직접 초기화 시도
            try:
                self.bm25_retriever = BM25Retriever(docs=self.documents)
                self.bm25_retriever.k = self.bm25_k
            except Exception as e2:
                self.bm25_retriever = None
        except Exception as e:
            self.bm25_retriever = None

    def _build_vector(self):
        """Vector Retriever 생성"""
        try:
            # VectorStore 상태 확인
            test_search = self.vectorstore.similarity_search("테스트", k=1)

            self.vector_retriever = self.vectorstore.as_retriever(
                search_type="similarity",
                search_kwargs={"k": self.vector_k}
```

```
            )
        except Exception as e:
            self.vector_retriever = None

    def dense_search(self, query: str, metadata_filter: Dict = None) ->
List[Tuple[any, float]]:
        """Dense 검색 (임베딩 기반)"""
        try:
            if self.vector_retriever:
                if metadata_filter:
                    # 메타데이터 필터 적용
                    docs = self.vectorstore.similarity_search(
                        query,
                        k=self.vector_k,
                        filter=metadata_filter
                    )
                else:
                    docs = self.vector_retriever.invoke(query)

                results = [(doc, 1.0) for doc in docs]
                return results
            return []
        except Exception as e:
            return []

    def bm25_search(self, query: str) -> List[Tuple[any, float]]:
        """BM25 검색 (키워드 기반)"""
        try:
            if self.bm25_retriever:
                docs = self.bm25_retriever.invoke(query)
                results = [(doc, 1.0) for doc in docs]
                return results
            return []
        except Exception as e:
            return []

    def retrieve(self, queries: List[str], metadata_filter: Dict = None) ->
Dict[str, List[Tuple[any, float]]]:
        """모든 검색 전략 실행"""
        all_results = {
            'dense': [],
            'bm25': []
        }

        for query in queries:
            all_results['dense'].extend(self.dense_search(query, metadata_filter))
            all_results['bm25'].extend(self.bm25_search(query))

        return all_results

    def get_ensemble(self, query: str) -> List[any]:
        """Ensemble 검색 (가중치 적용)"""
        if not RETRIEVERS_AVAILABLE or EnsembleRetriever is None:
            return self.dense_search(query)
```

```
        try:
            retrievers = []
            weights = []

            if self.bm25_retriever:
                retrievers.append(self.bm25_retriever)
                weights.append(self.bm25_weight)

            if self.vector_retriever:
                retrievers.append(self.vector_retriever)
                weights.append(self.vector_weight)

            if not retrievers:
                return []

            # 가중치 정규화
            total_weight = sum(weights)
            weights = [w / total_weight for w in weights]

            # LangChain의 EnsembleRetriever 사용
            ensemble = EnsembleRetriever(
                retrievers=retrievers,
                weights=weights
            )

            docs = ensemble.invoke(query)
            return docs

        except Exception as e:
            return []
```

특징:

- 하이브리드 검색: BM25(기본 weight=0.4) + 벡터(기본 weight=0.6)를 결합해 정밀도와 재현성 균형
- 메타데이터 필터 지원: 지역/전국 필터 등 `metadata_filter`가 적용되어 지역 기반 검색 가능
- 안정성 지향 초기화: BM25/Vector retriever 생성 실패 시 폴백 로직으로 가능한 리트리버를 사용
- LangChain `EnsembleRetriever`를 이용한 가중치 기반 통합(파라미터로 k 및 가중치 조정 가능)

4.4 ReciprocalRankFusion (순위 통합)

파일: `src/advanced_rag.py` (Line 315-370)

목적: 여러 검색 결과를 랭킹 기반으로 통합

코드:

```
class ReciprocalRankFusion:
    """여러 검색 결과를 랭킹 기반으로 통합"""
```

```
    def __init__(self, k: int = 60):
        self.k = k
    def fuse(self, results_dict: Dict[str, List[Tuple[any, float]]], top_k: int =
10) -> List[any]:
        doc_scores = {}
        for method, results in results_dict.items():
            for rank, (doc, score) in enumerate(results, 1):
                doc_id = doc.metadata.get('policy_id', id(doc))
                rrf_score = 1.0 / (self.k + rank)
                if doc_id not in doc_scores:
                    doc_scores[doc_id] = {'doc': doc, 'score': 0}
                doc_scores[doc_id]['score'] += rrf_score
        sorted_docs = sorted(doc_scores.items(), key=lambda x: x[1]['score'],
reverse=True)
        final_docs = [item[1]['doc'] for item in sorted_docs[:top_k]]
        return final_docs
```

특징:

- 다중 검색 결과 통합: 여러 쿼리와 검색 방법에서 얻은 결과를 순위 기반으로 결합
- RRF 스코어링: 각각의 결과 순위에 대해 `1/(k + rank)`를 합산하여 안정적인 상위 문서 선정
- 기본 파라미터: 클래스는 `k=60`를 사용하며, 파이프라인에서는 `top_k=20`으로 최종 선택(필요시 조정 가능)
- 중복 문서 병합과 순위 안정성 제공

4.5 ConversationMemory (대화 기록)

파일: `src/advanced_rag.py` (Line 375-420)

목적: 최근 대화 기록을 컨텍스트로 제공

코드:

```
@dataclass
class ConversationMemory:
    messages: List[Dict] = field(default_factory=list)
    max_history: int = 10
    def add_message(self, role: str, content: str):
        self.messages.append({"role": role, "content": content, "timestamp":
datetime.now().isoformat()})
        if len(self.messages) > self.max_history * 2:
            self.messages = self.messages[-self.max_history * 2:]
    def get_context(self) -> str:
        if not self.messages:
            return "이전 대화 없음"
        context_parts = []
        for msg in self.messages[-6:]:
            role = "사용자" if msg['role'] == 'user' else "AI"
            context_parts.append(f"{role}: {msg['content']}")
        return "\n".join(context_parts)
```

```
    def clear(self):
        self.messages.clear()
```

특징:

- 대화 맥락 유지: 최근 3턴(=최대 6개 메시지)을 기본으로 보관하여 컨텍스트 제공
- 크기 제어: `max_history`로 보관 길이 조정 가능, 초과 시 FIFO로 자동 정리
- 문자열 컨텍스트 반환: LLM 입력용으로 적합한 포맷(역할 라벨 포함)으로 변환

4.6 AdvancedRAGPipeline (통합 파이프라인)

파일: `src/advanced_rag.py` (Line 425-640)

목적: 모든 컴포넌트를 통합하여 전체 파이프라인 실행

핵심 메서드: `query()`

코드:

```
def query(self, user_query: str) -> Dict:
    """전체 파이프라인 실행"""

    # 1. Router: 질문 검증 및 정제
    if self.router:
        route_result = self.router.route(user_query)
        if not route_result['is_valid']:
            return {
                "answer": "음… 이해를 못 했어🥹. 한 번만 다시 얘기해줘!",
                "documents": [],
                "metadata": route_result
            }
        query = route_result['refined_query']
    else:
        query = user_query

    # 2. Region Filter: 지역 정보 추출 및 필터 생성
    metadata_filter = None
    region_info = None
    if self.region_filter:
        region_info = self.region_filter.detect_region(query)
        metadata_filter = self.region_filter.build_filter(region_info)

    # 3. Multi-Query: 다중 쿼리 생성
    if self.multi_query:
        queries = self.multi_query.generate(query)
    else:
        queries = [query]

    # 4. Ensemble Retriever: 다중 검색 (메타데이터 필터 적용)
    if self.ensemble:
        search_results = self.ensemble.retrieve(queries, metadata_filter)
```

```
    else:
        if metadata_filter:
            docs_with_score = self.vectorstore.similarity_search_with_score(query,
k=5, filter=metadata_filter)
        else:
            docs_with_score = self.vectorstore.similarity_search_with_score(query,
k=5)
        search_results = {'dense': docs_with_score}

    # 5. RRF: 검색 결과 통합
    if self.rrf:
        docs = self.rrf.fuse(search_results, top_k=20)
    else:
        docs = [doc for doc, score in search_results['dense']]

    # 6. Region Filter: 지역 기반 후처리 필터링
    if self.region_filter and region_info:
        docs = self.region_filter.filter_documents(docs, region_info)

    # 7. Memory: 대화 맥락 가져오기
    if self.memory:
        context = self.memory.get_context()
    else:
        context = "이전 대화 없음"

    # 8. LLM: 최종 답변 생성 (정책명 중복 없이 최대 3개만)
    seen_titles = set()
    unique_docs = []
    for doc in docs:
        title = doc.metadata.get('정책명', '제목 없음')
        if title not in seen_titles:
            seen_titles.add(title)
            unique_docs.append(doc)
        if len(unique_docs) >= 3:
            break
    docs_text = "\\n\\n".join([
        f"[정책 {i+1}] {doc.metadata.get('정책명', '제목 없
음')}\\n{doc.page_content[:500]}"
        for i, doc in enumerate(unique_docs)
    ])
    try:
        response = self.answer_prompt | self.llm | StrOutputParser()
        answer = response.invoke({
            "context": context,
            "documents": docs_text,
            "query": user_query
        })
        summary_response = self.summary_prompt | self.llm | StrOutputParser()
        summary = summary_response.invoke({"answer": answer})
        if self.memory:
            self.memory.add_message("user", user_query)
            self.memory.add_message("assistant", answer)
        return {
            "answer": answer,
```

```
            "summary": summary,
            "documents": docs,
            "metadata": {
                "queries": queries,
                "num_docs_retrieved": len(docs),
                "has_context": bool(self.memory and self.memory.messages),
                "region_filter": metadata_filter
            }
        }
    except Exception as e:
        return {
            "answer": "답변 생성 중 오류 발생",
            "documents": [],
            "metadata": {"error": str(e)}
        }
```

특징:

- 단계별 통합 파이프라인: Router → Region Filter → Multi-Query → Ensemble Retriever → RRF → Region 후처리 → Memory → LLM 순으로 실행
- 모듈화 및 가변성: 초기화 시 각 컴포넌트(router, multi_query, ensemble, rrf, memory, region_filter)를 켜/끄기 가능
- 출력 제한 및 중복 제거: 문서 중복 제거 후 기본적으로 최대 3건의 정책을 요약·출력(설정 변경 가능)
- 풍부한 반환 메타데이터: 쿼리 목록, 검색된 문서 수, 컨텍스트 유무, 적용된 메타필터 등 반환
- 오류 폴백: 각 단계 실패 시 단계별 폴백 로직으로 안정적 응답 제공

---

5. 벡터 데이터베이스 연동

5.1 ChromaDB 초기화

파일: `src/advanced_rag.py` (Line 548-640)

함수: `initialize_rag_pipeline()`

코드:

```
def initialize_rag_pipeline(vectordb_path: str = None, api_key: str = None):
    """
    Streamlit에서 사용할 수 있는 RAG 파이프라인 초기화 함수

    Args:
        vectordb_path: VectorDB 경로 (None이면 자동 계산)
        api_key: OpenAI API Key (None이면 환경변수 사용)

    Returns:
        AdvancedRAGPipeline: 초기화된 파이프라인 객체
    """
    # API Key 설정
    if api_key:
        os.environ['OPENAI_API_KEY'] = api_key
```

```
    else:
        api_key = os.getenv('OPENAI_API_KEY')

    if not api_key:
        raise ValueError('OPENAI_API_KEY가 설정되지 않았습니다.')

    # LLM 및 임베딩 초기화
    llm = ChatOpenAI(
        model="gpt-4o-mini",
        temperature=0.0,
        api_key=api_key
    )

    embeddings = OpenAIEmbeddings(
        model="text-embedding-3-small",
        api_key=api_key
    )

    # VectorDB 경로 설정
    if vectordb_path is None:
        vectordb_path = os.path.join(os.getcwd(), "data", "vectordb")

    if not os.path.exists(vectordb_path):
        raise FileNotFoundError(f"VectorDB 경로가 존재하지 않습니다:
{vectordb_path}")

    # VectorStore 로드
    vectorstore = Chroma(
        collection_name="youth_policies",
        embedding_function=embeddings,
        persist_directory=vectordb_path
    )

    # 문서 로드 (BM25를 위해 필요)
    all_docs = vectorstore.get()

    if not all_docs or not all_docs.get('documents'):
        raise ValueError("VectorDB에 문서가 없습니다.")

    documents = []
    for i, doc_text in enumerate(all_docs['documents']):
        if doc_text and doc_text.strip():
            metadata = all_docs['metadatas'][i] if 'metadatas' in all_docs else {}
            documents.append(Document(page_content=doc_text, metadata=metadata))

    # RAG 파이프라인 생성
    rag = AdvancedRAGPipeline(
        documents=documents,
        vectorstore=vectorstore,
        llm=llm,
        enable_router=True,
        enable_multi_query=True,
        enable_ensemble=True,
        enable_rrf=True,
```

```
        enable_memory=True,
        enable_region_filter=True,
        bm25_k=5,
        vector_k=10,
        bm25_weight=0.4,
        vector_weight=0.6
    )

    return rag
```

특징:

- 유연한 초기화: `api_key` 파라미터 또는 환경변수 `OPENAI_API_KEY`로 설정 가능
- LLM/임베딩 설정: 기본 LLM은 `gpt-4o-mini`, 임베딩은 `text-embedding-3-small`으로 초기화(필요시 변경 가능)
- VectorDB 로딩 및 검증: 지정 경로가 존재하는지 확인하고, DB에서 문서와 메타데이터를 읽어 `Document` 객체로 변환
- 기본 구성: 모든 핵심 컴포넌트(router, multi_query, ensemble, rrf, memory, region_filter)를 활성화한 상태로 반환(필요시 플래그로 비활성화 가능)

5.2 벡터 DB 구축

파일: `notebooks/build_vectordb.py` (Line 1-572)

주요 함수:

1) 데이터 로드

```
def load_preprocessed_data(filepath):
    """
    전처리된 JSON 데이터 로드

    Args:
        filepath: JSON 파일 경로

    Returns:
        list: 정책 데이터 리스트
    """

    with open(filepath, 'r', encoding='utf-8') as f:
        data = json.load(f)

    return data
```

2) 텍스트 생성

```
def create_policy_text(policy):
    """
    정책 데이터를 임베딩을 위한 텍스트로 변환
    중요도 순으로 배치하여 검색 품질 향상

    Args:
        policy: 정책 딕셔너리

    Returns:
        str: 결합된 텍스트
    """
    # 주요 필드들을 결합하여 검색 가능한 텍스트 생성 (중요도 순)
    text_parts = []

    # 1. 가장 중요: 정책명과 분야
    if policy.get('정책명'):
        text_parts.append(f"정책명: {policy['정책명']}")

    if policy.get('대분류'):
        text_parts.append(f"대분류: {policy['대분류']}")

    if policy.get('중분류'):
        text_parts.append(f"중분류: {policy['중분류']}")

    # 2. 정책 설명 (핵심 내용)
    if policy.get('정책설명'):
        text_parts.append(f"정책설명: {policy['정책설명']}")

    # 3. 지원내용 (길이 제한 없음 - 중요한 정보)
    if policy.get('지원내용'):
        text_parts.append(f"지원내용: {policy['지원내용']}")

    if policy.get('정책키워드'):
        text_parts.append(f"키워드: {policy['정책키워드']}")

    # 4. 지역 정보 (검색 정확도 향상)
    if policy.get('지역'):
        # 지역 정보가 길 수 있으므로 적절히 포함
        region = policy['지역']
        # 쉼표로 구분된 지역을 간단히 처리
        if len(region) > 500:
            # 너무 길면 앞부분만 (전국 정책일 가능성)
            region = region[:500] + "..."
        text_parts.append(f"적용지역: {region}")

    if policy.get('지역범위'):
        region_level = policy['지역범위']

    # 지역 범위(전국/지역)을 자연어 라벨로 추가
        if region_level == '전국':
            text_parts.append("지역범위: 전국 (모든 지역 적용)")
        else:
            text_parts.append(f"지역범위: 지역 적용 (일부 지역 적용)")
```

```
        # 5. 자격 조건 (상세)
        if policy.get('추가자격조건'):
            # 길이 제한 확대 (500자)
            qual = policy['추가자격조건'][:500]
            text_parts.append(f"자격조건: {qual}")

        # 6. 자격 요건 (한글 변환된 필드)
        if policy.get('취업상태'):
            job_status = policy['취업상태']
            # 자연어 표현 추가 (검색 향상)
            natural_terms = []
            if '미취업자' in job_status:
                natural_terms.append('실업자, 구직자, 백수, 취업준비생')
            if '재직자' in job_status:
                natural_terms.append('직장인, 근로자')
            if '창업자' in job_status or '예비' in job_status:
                natural_terms.append('창업준비, 사업자')

            full_status = f"취업상태: {job_status}"
            if natural_terms:
                full_status += f" ({', '.join(natural_terms)})"
            text_parts.append(full_status)

        if policy.get('학력요건'):
            edu_req = policy['학력요건']
            # 자연어 표현 추가
            natural_edu = []
            if '고졸' in edu_req or '고교' in edu_req:
                natural_edu.append('고등학교')
            if '대학' in edu_req or '대졸' in edu_req:
                natural_edu.append('대학교, 학사')
            if '석박사' in edu_req:
                natural_edu.append('대학원')

            full_edu = f"학력: {edu_req}"
            if natural_edu:
                full_edu += f" ({', '.join(natural_edu)})"
            text_parts.append(full_edu)

        if policy.get('전공요건'):
            major_req = policy['전공요건']
            # 자연어 표현 추가
            natural_major = []
            if '공학' in major_req:
                natural_major.append('이공계, 기술, IT, 공대')
            if '상경' in major_req:
                natural_major.append('경영, 경제, 회계')
            if '예체능' in major_req:
                natural_major.append('예술, 체육, 음악, 미술')

            full_major = f"전공: {major_req}"
            if natural_major:
                full_major += f" ({', '.join(natural_major)})"
```

```
            text_parts.append(full_major)

        if policy.get('특화분야'):
            special = policy['특화분야']
            # 자연어 표현 추가
            natural_special = []
            if '여성' in special:
                natural_special.append('여성청년, 경력단절여성')
            if '장애인' in special:
                natural_special.append('장애청년')
            if '한부모' in special:
                natural_special.append('싱글맘, 싱글대디, 미혼모, 미혼부')
            if '기초생활수급자' in special:
                natural_special.append('저소득층, 차상위계층')
            if '중소기업' in special:
                natural_special.append('중견기업, 스타트업')

            full_special = f"특화분야: {special}"
            if natural_special:
                full_special += f" ({', '.join(natural_special)})"
            text_parts.append(full_special)

        if policy.get('정책제공방법'):
            text_parts.append(f"제공방법: {policy['정책제공방법']}")

        if policy.get('소득조건'):
            income = policy['소득조건']
            # 자연어 표현 추가
            natural_income = []
            if '무관' in income:
                natural_income.append('소득제한없음, 누구나')
            if '연소득' in income:
                natural_income.append('소득기준, 소득제한')

            full_income = f"소득조건: {income}"
            if natural_income:
                full_income += f" ({', '.join(natural_income)})"
            text_parts.append(full_income)

        if policy.get('혼인상태'):
            text_parts.append(f"혼인상태: {policy['혼인상태']}")

        # 7. 연령 제한
        min_age = policy.get('지원최소연령', '0')
        max_age = policy.get('지원최대연령', '0')
        if min_age != '0' or max_age != '0':
            age_info = f"대상연령: {min_age}세 ~ {max_age}세"
            text_parts.append(age_info)

        # 8. 지원금액
        min_amount = policy.get('최소지원금액', '0')
        max_amount = policy.get('최대지원금액', '0')
        if min_amount != '0' or max_amount != '0':
            amount_info = f"지원금액: {min_amount}원 ~ {max_amount}원"
```

```
            text_parts.append(amount_info)

        if policy.get('기타지원조건'):
            other_cond = policy['기타지원조건'][:200]
            text_parts.append(f"기타조건: {other_cond}")

        # 9. 신청 및 선정 방법
        if policy.get('신청기간구분'):
            text_parts.append(f"신청기간: {policy['신청기간구분']}")

        if policy.get('사업기간구분'):
            text_parts.append(f"사업기간: {policy['사업기간구분']}")

        if policy.get('신청방법'):
            method = policy['신청방법'][:200]  # 너무 길면 제한
            text_parts.append(f"신청방법: {method}")

        if policy.get('선정방법'):
            selection = policy['선정방법'][:200]
            text_parts.append(f"선정방법: {selection}")

        if policy.get('제출서류'):
            docs = policy['제출서류'][:200]
            text_parts.append(f"제출서류: {docs}")

        # 10. 참여제외대상 (중요 정보)
        if policy.get('참여제외대상'):
            exclusion = policy['참여제외대상'][:300]
            text_parts.append(f"제외대상: {exclusion}")

        # 텍스트가 비어있으면 최소한의 정보라도 포함
        if not text_parts:
            text_parts.append(f"청년정책")

        # 텍스트 결합 및 정제
        full_text = "\n".join(text_parts)

        # 중복 공백 제거 및 정리
        full_text = " ".join(full_text.split())

        return full_text
```

3) 임베딩 생성

```
def get_embedding(text, model="text-embedding-3-small"):
    """
    OpenAI API를 사용하여 텍스트 임베딩 생성

    Args:
        text: 임베딩할 텍스트
        model: 사용할 임베딩 모델
```

```
            - text-embedding-3-small (1536차원, 빠름, 저렴)
            - text-embedding-3-large (3072차원, 느림, 고품질, 고비용)

    Returns:
        list: 임베딩 벡터
    """
    # 텍스트 정제
    text = text.replace("\n", " ").strip()

    # 빈 텍스트 체크
    if not text or len(text) < 3:
        text = "정책 정보"

    # 너무 긴 텍스트는 잘라내기
    # text-embedding-3-small: 최대 8191 토큰 (~32,000자)
    # 안전을 위해 8000자로 제한
    if len(text) > 8000:
        text = text[:8000]

    # API 호출 재시도 로직 (Rate limit 대응)
    max_retries = 3
    for attempt in range(max_retries):
        try:
            response = client.embeddings.create(input=[text], model=model)
            return response.data[0].embedding
        except Exception as e:
            if attempt < max_retries - 1:
                import time
                wait_time = (attempt + 1) * 2
                time.sleep(wait_time)
            else:
                raise e
```

4) ChromaDB 저장

```
def build_chromadb(policies, db_path="../data/vectordb"):
    """
    ChromaDB 벡터 데이터베이스 구축

    Args:
        policies: 정책 데이터 리스트
        db_path: DB 저장 경로
    """
    # DB 디렉토리 생성
    current_dir = os.path.dirname(os.path.abspath(__file__))
    project_root = os.path.dirname(current_dir)
    db_full_path = os.path.join(project_root, "data", "vectordb")
    os.makedirs(db_full_path, exist_ok=True)

    # ChromaDB 클라이언트 초기화
    chroma_client = chromadb.PersistentClient(path=db_full_path)
```

```
    # 기존 컬렉션 삭제 (있으면)
    try:
        chroma_client.delete_collection(name="youth_policies")
    except:
        pass

    # 물리적 파일 정리 (세그먼트 폴더 삭제)
    import shutil
    for item in os.listdir(db_full_path):
        item_path = os.path.join(db_full_path, item)
        # UUID 형식의 폴더만 삭제 (chroma.sqlite3는 유지)
        if os.path.isdir(item_path) and '-' in item:
            try:
                shutil.rmtree(item_path)
            except:
                pass

    # 새 컬렉션 생성
    collection = chroma_client.create_collection(
        name="youth_policies",
        metadata={"description": "온통청년 정책 데이터"}
    )

    # 배치 임베딩 처리를 위한 변수
    embedding_batch_size = 20  # OpenAI API 배치 제한
    db_batch_size = 50  # DB 저장 배치

    all_policy_texts = []
    all_metadatas = []
    all_ids = []

    failed_count = 0

    # 1단계: 모든 정책 텍스트 생성
    for idx, policy in enumerate(policies, 1):
        try:
            policy_text = create_policy_text(policy)
            all_policy_texts.append(policy_text)

            all_metadatas.append({
                '정책명': policy.get('정책명', ''),
                '대분류': policy.get('대분류', ''),
                '중분류': policy.get('중분류', ''),
                '주관기관명': policy.get('주관기관명', ''),
                '운영기관명': policy.get('운영기관명', ''),
                '등록기관명': policy.get('등록기관명', ''),
                '상위기관명': policy.get('상위기관명', ''),
                '상위등록기관명': policy.get('상위등록기관명', ''),
                '신청URL': policy.get('신청URL', ''),
                '참고URL1': policy.get('참고URL1', ''),
                '정책키워드': policy.get('정책키워드', ''),
                # 신청 관련
                '신청기간': policy.get('신청기간', ''),
```

```
                '신청방법': policy.get('신청방법', ''),
                '제출서류': policy.get('제출서류', ''),
                # 사업 기간
                '사업시작일': policy.get('사업시작일', ''),
                '사업종료일': policy.get('사업종료일', ''),
                # 심사·선정
                '선정방법': policy.get('선정방법', ''),
                # 자격 관련
                '추가자격조건': policy.get('추가자격조건', ''),
                '참여제외대상': policy.get('참여제외대상', ''),
                '지원최소연령': policy.get('지원최소연령', '0'),
                '지원최대연령': policy.get('지원최대연령', '0'),
                # 지원금 관련
                '최소지원금액': policy.get('최소지원금액', '0'),
                '최대지원금액': policy.get('최대지원금액', '0'),
                '기타지원조건': policy.get('기타지원조건', ''),
                # 한글 변환된 필드들
                '재공기관그룹': policy.get('재공기관그룹', ''),
                '정책제공방법': policy.get('정책제공방법', ''),
                '정책승인상태': policy.get('정책승인상태', ''),
                '신청기간구분': policy.get('신청기간구분', ''),
                '사업기간구분': policy.get('사업기간구분', ''),
                '혼인상태': policy.get('혼인상태', ''),
                '소득조건': policy.get('소득조건', ''),
                '전공요건': policy.get('전공요건', ''),
                '취업상태': policy.get('취업상태', ''),
                '학력요건': policy.get('학력요건', ''),
                '특화분야': policy.get('특화분야', ''),
                '지역': policy.get('지역', ''),
                '지역범위': policy.get('지역범위', ''),
            })
            all_ids.append(f"policy_{idx}")

        except Exception as e:
            failed_count += 1
            all_policy_texts.append("청년정책")
            all_metadatas.append({})
            all_ids.append(f"policy_{idx}")

    # 2단계: 배치 임베딩 생성
    all_embeddings = []

    for i in range(0, len(all_policy_texts), embedding_batch_size):
        batch_texts = all_policy_texts[i:i+embedding_batch_size]
        try:
            batch_embeddings = get_embeddings_batch(batch_texts)
            all_embeddings.extend(batch_embeddings)
        except Exception as e:
            # 폴백: 개별 임베딩
            for text in batch_texts:
                try:
                    emb = get_embedding(text)
                    all_embeddings.append(emb)
                except:
```

```
                    all_embeddings.append([0] * 1536)  # 빈 벡터
                    failed_count += 1

    # 3단계: DB에 배치 저장
    for i in range(0, len(all_policy_texts), db_batch_size):
        batch_docs = all_policy_texts[i:i+db_batch_size]
        batch_metas = all_metadatas[i:i+db_batch_size]
        batch_ids = all_ids[i:i+db_batch_size]
        batch_embs = all_embeddings[i:i+db_batch_size]

        collection.add(
            documents=batch_docs,
            metadatas=batch_metas,
            ids=batch_ids,
            embeddings=batch_embs
        )

    return collection
```

특징:

- 임베딩/저장 배치: 임베딩 배치 크기(`embedding_batch_size`)는 기본 20, DB에 저장하는 배치(`db_batch_size`)는 기본 50으로 설정되어 효율적 처리
- API 제한 대응: OpenAI 임베딩 호출에 재시도 로직을 적용하고, 실패 시 개별 폴백 또는 빈 벡터로 대체
- 메타데이터 정교화: 정책별 다수 메타필드를 매핑하여 검색 시 필터링 및 표시 정보로 활용
- 안전한 초기화 및 정리: 기존 컬렉션 삭제 및 물리적 세그먼트 폴더 정리 로직 포함
- 진행률/오류 집계: 실패 건수 집계 및 처리 상태 확인 가능
