ChatGPT

SafeGuard AI 에이전트형 RAG 설계

SafeGuard AI는 **스마트 카메라 이미지**, **IoT 센서 메타데이터**, **PDF 작업 지시서** 등의 이기종 데이터를 통합 분석하여 산업 현장의 위험을 판단하고 대응 방안을 제시하는 시스템입니다. 이를 위해 **에이전트 기반의 검색 증강 생성(RAG)** 아키텍처를 LangChain과 Hugging Face 모델로 구현합니다. 아래에서는 폴더 구조와 구성 요소, 동작 흐름, 사용 기술 스택 및 향후 확장 방향을 설계 관점에서 설명합니다.

폴더 및 코드베이스 구조

프로젝트는 기능별로 모듈을 분리하여 유지보수성과 확장성을 높입니다. 주요 디렉토리 구조는 다음과 같습니다:

- `data/` – 정적 자원 디렉토리로, PDF 작업지시서 원본 파일과 벡터 인덱스 파일 등을 저장합니다. 예를 들어 초기 세팅 시 여기에 PDF를 넣고, 임베딩 완료 후 생성된 FAISS 인덱스 파일(`index.faiss` 등)을 저장할 수 있습니다.
- `ingestion/` – **문서 로딩 및 임베딩 파이프라인** 모듈입니다.
- `pdf_parser.py`: PyMuPDF 또는 pdfminer.six를 활용하여 PDF 문서를 텍스트로 추출하는 로더 클래스 구현. 페이지 구분, 이미지 캡션 등 필요한 메타데이터 추출 로직 포함.
- `text_splitter.py`: 긴 문서를 잘게 쪼개기 위한 텍스트 분할기(예: LangChain의 `RecursiveCharacterTextSplitter`) 설정. chunk 크기, 중첩 정도 등을 정의.
- `embedder.py`: Hugging Face의 사전훈련 임베딩 모델을 불러와 텍스트 조각을 벡터로 변환하는 모듈. SentenceTransformer 계열 모델 등을 사용.
- `index_builder.py`: 벡터화된 임베딩을 **벡터 데이터베이스**에 색인하는 스크립트. FAISS를 사용할 경우 이 모듈에서 index 생성 및 지속화, OpenSearch를 사용할 경우 REST API로 인덱싱.
- `vectorstore/` – **벡터 DB 연동** 모듈. 벡터스토어 초기화 및 쿼리 인터페이스를 캡슐화합니다.
- `faiss_store.py`: FAISS 인덱스를 생성/로드하고 쿼리하는 클래스 (`FAISS.from_documents` 등을 래핑).
- `opensearch_store.py`: OpenSearch KNN 인덱스 사용 시, OpenSearch Python 클라이언트를 통해 색인 생성, 유사도 검색 쿼리 수행 기능 구현. LangChain의 `OpenSearchVectorSearch` 래퍼를 내부적으로 활용합니다.
- `agent/` – **에이전트 및 툴체인** 관련 코드.
- `tools.py`: LangChain **툴** 정의 모듈. 예를 들어 작업지시서 검색용 `DocumentSearchTool` 클래스 (벡터 DB에서 관련 문서 조각을 검색)과 필요시 위험도 평가용 `RiskAssessmentTool` 함수 등을 구현. 각 툴에는 `name`, `description`, `func`가 정의되어 에이전트가 호출 가능하도록 합니다.
- `agent_manager.py`: 에이전트 초기화 및 실행 모듈. LangChain의 에이전트 세팅(LLM, 툴 목록, 프롬프트 템플릿 등)을 구성합니다. 에이전트의 System 프롬프트에 역할 지시 (`예: 당신은 산업안전 도우미입니다...`)와 응답 형식 지침을 포함하고, 사용 가능한 툴들을 등록합니다.
- `chains.py` (선택사항): 복잡한 작업을 위해 체인을 미리 정의할 경우 사용. 예를 들어 Retrieval QA 체인이나 멀티스텝 체인을 구성하여 에이전트에서 활용할 수도 있습니다.
- `models/` – **Hugging Face 모델 로딩** 모듈.
- `hf_model.py`: Hugging Face Transformer 모델을 로컬로 로드하고 LangChain에서 사용할 수 있도록 래핑합니다. 토크나이저 설정, 장치(gpu) 할당, `HuggingFacePipeline` 객체 생성 등을 수행합니다. 예를 들어 `AutoModelForCausalLM.from_pretrained()`로 모델을 불러와 generation 파이프라인을 구성합니다.

- `embedding_model.py`: 문서 임베딩에 사용하는 Transformer 모델 로딩 (예: `SentenceTransformer` 모델). LangChain의 `HuggingFaceEmbeddings` 클래스를 사용하여 임베딩기 객체를 생성합니다.
- `api/` – **서비스 인터페이스** 모듈 (FastAPI 등).
- `main.py`: FastAPI 엔드포인트 정의 (예: `/analyze_event`). 요청이 들어오면 이벤트 메타데이터를 받아 에이전트를 호출하고, 생성된 대응 방안을 JSON/메시지 형태로 반환합니다.
- `schemas.py`: API 입출력 데이터 모델(Pydantic) 정의. 이벤트 정보(이미지 경로 또는 ID, 센서 데이터 등)와 응답 포맷(위험등급, 대응조치 리스트 등)을 스키마로 정의합니다.
- `app.py`: 앱 팩토리 및 에이전트 초기화 로직을 포함. 서버 기동 시 벡터 DB 및 모델을 로드하고 FastAPI 앱 인스턴스를 생성합니다.

이러한 구조를 통해 **문서 처리 (ingestion)** 단계와 **실시간 질의응답 (agent)** 단계를 명확히 분리하고, 향후 모듈 교체 (예: 벡터DB를 FAISS에서 OpenSearch로 교체 등)도 해당 디렉토리만 수정하여 유연하게 대응할 수 있습니다.

구성 요소별 설명

RAG 파이프라인 개념도 – 좌측은 PDF 작업지시서로부터 임베딩을 생성해 벡터DB에 색인(Indexing) 하는 단계이고, 우측은 이벤트로부터 질의를 만들어 관련 문서를 검색(Retrieval) 후 LLM이 최종 답변을 생성하는 단계이다.

위 그림처럼 RAG 시스템은 크게 **오프라인 색인 단계**와 **온라인 검색/생성 단계**로 나눌 수 있습니다. SafeGuard AI에서는 초기 설정 시 여러 작업 지시서 PDF를 임베딩하여 벡터DB에 저장해두고, 실시간으로 카메라/센서 이벤트가 발생할 때마다 해당 지식을 조회하여 대응합니다. 이제 각 구성 요소를 세부적으로 설명합니다.

문서 파서 및 임베딩 처리기

1) PDF 문서 파싱: 업로드된 작업 지시서(PDF 형식)를 효과적으로 처리하기 위해 PyMuPDF 등의 라이브러리를 사용합니다. PyMuPDF는 PDF에서 텍스트를 추출하는 **속도와 정확도** 측면에서 매우 우수하여, PDFplumber나 PyPDF2 등의 다른 오픈소스 대안보다 뛰어난 성능을 보여줍니다 [1]. 파서 모듈은 PDF의 각 페이지를 순회하며 텍스트 박스를 추출하고, 표나 리스트 같은 구조도 유지할 수 있도록 전처리합니다. 필요하다면 챕터/섹션 제목을 인식해 메타데이터로 저장하고, 페이지 번호도 각 문서 조각의 metadata에 기록합니다.

2) 텍스트 청크 분할: 추출된 문서 본문은 그대로 벡터화하기엔 길 수 있으므로, **텍스트 분할기**를 사용해 적절한 크기의 청크로 쪼갭니다. 예를 들어 LangChain의 `RecursiveCharacterTextSplitter`를 활용하여 한 청크당 최대 500자~1000자 정도로 분할하고, 인접 청크 간 50자 정도의 overlap을 두어 문맥이 끊기지 않도록 합니다. 이러한 분할을 통해 검색 및 LLM 입력 시 **컨텍스트 윈도우 제한**을 준수하고, 세밀한 검색 정확도를 높입니다.

3) 임베딩 벡터 변환: 각 텍스트 청크는 Hugging Face의 **사전훈련 임베딩 모델**을 통해 벡터로 변환합니다. 구체적으로는 `sentence-transformers` 라이브러리의 멀티리니어 임베딩 모델(예: `all-MiniLM-L6-v2`)을 많이 사용하며, LangChain의 `HuggingFaceEmbeddings` 클래스로 편리하게 래핑하여 이용합니다. 해당 모델은 텍스트의 의미를 384차원 등의 벡터로 표현하며, 유사한 내용일수록 벡터 간 코사인 유사도가 높아지도록 학습되어 있습니다. 다국어 작업지시서를 다루는 경우 **다국어 임베딩 모델** (예: `sentence-transformers/paraphrase-multilingual-MiniLM`)을 사용하거나, 한국어 문서에 특화된 모델을 선택할 수 있습니다.

4) 벡터DB 색인 저장: 임베딩된 벡터들과 함께 각 청크의 출처 문서명, 페이지 등 메타정보를 **벡터 데이터베이스**에 저장합니다. 이 단계에서는 두 가지 옵션을 고려할 수 있습니다: - **FAISS**: 페이스북 AI가 공개한 **경량 벡터 서치 라이브러리**로, 메모리 상에서 대규모 벡터들에 대한 근사 최근접 검색을 고속으로 수행합니다. 구현이 단순하고 임베딩 수천~수백만 규모까지도 효율적입니다. LangChain에서도 FAISS 연동을 기본 지원하며, Chroma 등과 함께 자주 사용되는 벡터 스토어입니다 [2]. FAISS는 별도 서버 없이 로컬 파일(`.index`) 형태로 저장할 수 있어, 초기 프로토타입이나 소규모 시스템에 적합합니다. - **OpenSearch**: Elasticsearch에서 포크되어 나온 **검색/분석 엔진**으로, 텍스트 검색뿐 아니라 벡터 검색 기능(KNN 서치)을 제공합니다. OpenSearch를 벡터DB로 활용하면, 대용량 데이터에 대해 **분산 환경**에서

확장성과 내구성을 갖춘 검색을 구현할 수 있습니다. OpenSearch는 벡터 임베딩을 인덱싱하고 유사도 검색을 지원하여, 키워드 매칭 이상의 **의미 기반 검색**을 가능하게 합니다 [3]. LangChain에는 `OpenSearchVectorSearch` 래퍼가 있어 OpenSearch에 임베딩을 저장하고 질의 시 유사 문서 조각을 가져오는 통합이 용이합니다 [4]. 즉, OpenSearch 백엔드를 사용하더라도 LangChain의 Retriever 인터페이스로 활용할 수 있습니다.

> **구현 참고:** 소규모 배포 단계에서는 설정이 간편한 FAISS를 우선 채택하고, 추후 데이터가 방대해지거나 서비스 형태로의 운영이 필요해지면 OpenSearch로 마이그레이션하는 방안을 고려합니다. 두 벡터 DB의 사용은 추상화되어 있기 때문에, `vectorstore` 모듈의 구현만 교체하면 다른 부분 코드는 영향을 받지 않도록 설계합니다.

LangChain Retriever 및 툴 설계

Retriever 설정: 색인된 벡터DB에 질의를 던져 관련 문서를 찾아주는 **리트리버(Retriever)**를 구성합니다. LangChain에서는 VectorStore로부터 `.as_retriever()` 메서드로 표준화된 Retriever 객체를 얻을 수 있습니다. 예를 들어 FAISS의 경우 `faiss_index.as_retriever(search_type="similarity", k=3)` 형태로 사용하여, 주어진 쿼리에 대해 **코사인 유사도 기준 상위 3개**의 문서 조각을 반환하도록 설정합니다 [5]. 이때 `search_type`이나 `filter` 옵션을 통해 필요한 경우 메타데이터 필터링(예: 작업 현장 종류, 문서 언어 등)도 가능합니다. Retriever는 **embedding 쿼리**를 내부에서 수행하므로, 사용자는 자연어로 질의를 던지기만 하면 벡터 DB에서 의미적으로 가장 관련 높은 문서들을 얻을 수 있습니다.

툴 설계: 에이전트가 Retriever를 활용하려면, 이를 **툴(Tool)** 형태로 노출해야 합니다. 툴은 에이전트가 호출할 수 있는 함수 인터페이스이며, 보통 **함수명**과 **설명(description)**으로 정의됩니다. SafeGuard AI에서 주요 툴은 다음과 같습니다:

- **문서검색 툴 (`DocumentSearchTool`)** – 작업지시서 벡터DB에서 **관련 정보 조회**를 수행합니다. 이 툴의 `func` 구현은 위에서 설정한 Retriever를 호출하여 문서 조각들의 내용을 반환합니다. 에이전트가 이 툴을 사용할 때 혼동을 줄이기 위해, description에 "이상 상황과 관련된 작업 지시서 내용을 검색" 등의 설명을 달아둡니다. 툴 함수는 검색 결과 청크들의 텍스트를 하나의 문자열로 묶어 반환하거나, 결과가 너무 길면 상위 몇 개만 요약하여 반환합니다. (너무 많은 텍스트를 한꺼번에 LLM에 주면 맥락이 흐려질 수 있으므로, 필요시 요약 또는 **컨텍스트 압축** 기법을 활용합니다.)

- **위험도 평가 툴 (`RiskAssessmentTool`)** – 이 툴은 현재 상황의 위험도를 산정하는 기능입니다. 두 가지 구현 접근이 있습니다: (a) 룰기반 함수 – 예를 들어 입력으로 받은 상황 메타데이터(온도, 압력 수치 등)나 탐지된 이벤트 종류에 따라 미리 정의된 위험 등급을 반환하는 함수. (b) LLM 프롬프트 체인 – 상황 설명과 관련 문서 내용을 입력으로 받아 "위험도를 높음/중간/낮음 중 하나로 분류하고 근거를 1문장으로 제시"하도록 LLM에 요청하는 체인. **설계상** (a)는 결정론적이어서 일관성은 있으나 유연성이 낮고, (b)는 유연하나 LLM 출력의 신뢰도 문제가 있을 수 있습니다. SafeGuard AI에서는 초기에는 에이전트 자체가 종합 답변을 생성할 때 위험도까지 함께 기술하게 하고, 필요하면 후에 (b) 방식의 체인을 보조적으로 활용할 수 있습니다.

- **대응 조치 생성 툴** – 보통 대응 방안 제안은 최종 답변 생성의 일부로 LLM이 수행하므로 별도 툴로 구현하지는 않습니다. 그러나 특정 상황에서는 정형화된 체크리스트나 절차를 생성하는 함수를 툴로 만들어 둘 수 있습니다. 예를 들어 "소화기 작동 절차 출력" 같은 툴을 미리 만들어 두면, 에이전트는 화재 상황시 이 툴을 호출해 표준 절차를 가져온 뒤 자신의 답변과 결합할 수 있습니다. (이런 툴은 시스템에 내장된 템플릿이나 DB에서 정보를 가져오는 용도로 구현 가능합니다.)

  툴체인 구성: 위와 같은 툴들을 등록하면, 에이전트는 필요에 따라 여러 툴을 **연쇄적으로 호출**하여 문제를 해결합니다. 예를 들어 에이전트 프롬프트에는 "너는 산업 안전 에이전트이다. 상황에 따라 적절한 도구를 사용해 답을 찾아라."라고 지시하고, 툴로 `DocumentSearchTool`과 `RiskAssessmentTool`을 주었다면, 에이전트는 먼저 DocumentSearchTool을 사용해 관련 지식을 찾고, 이어서 RiskAssessmentTool을 사용해 등급을 분류한 뒤, 최종적으로 자체 답변을 구성하

는 식의 체계를 갖출 수 있습니다. (툴 사용 여부와 순서는 에이전트가 **자율적으로 결정**하지만, prompt 설계를 통해 필요한 경우 특정 순서를 유도할 수 있습니다.)

에이전트 및 Toolchain 구성

LLM 에이전트 구성: LangChain의 에이전트 프레임워크를 활용하여 SafeGuard AI의 **브레인**을 구성합니다. 에이전트의 핵심은 **대형 언어모델(LLM)**이며, 여기에 앞서 정의한 툴들을 연결합니다. Hugging Face의 오픈소스 모델을 LLM으로 사용하므로, `agent_manager.py`에서 HuggingFace 모델을 불러와 LangChain의 `AgentExecutor` 또는 `initialize_agent`를 통해 툴들을 등록한 에이전트를 생성합니다. 에이전트의 프롬프트 템플릿에는 시스템 역할로 다음과 같은 지침을 포함합니다:

- **역할/목표**: "당신은 산업 현장의 안전 전문가 AI입니다. 주어진 카메라/센서 이벤트 정보를 바탕으로, 필요한 경우 작업지시서 내용을 찾아보고, 현재 상황의 위험 수준과 그 근거, 그리고 권장 대응 조치를 제시하세요." 등의 내용으로, 에이전트의 최종 답변 형식을 간단히 지정합니다 (예: "위험도: 높음, 조치: ..." 식으로).
- **도구 사용 지시**: "질문에 답하기 위해 필요하면 다음 도구들을 사용할 수 있다:"라고 안내하고, 각 툴의 이름과 description을 나열합니다. 에이전트는 이 description을 참고하여 언제 어떤 툴을 쓸지 판단합니다. 예를 들어 DocumentSearchTool의 설명에 "관련 작업 지시서 내용을 검색"이라고 되어 있으므로, 에이전트는 질문에 바로 답을 모를 때 이 툴을 먼저 시도하게 됩니다.
- **포맷 지시**: LangChain의 ReAct 에이전트 패턴을 따르면, 프롬프트에 Observation, Thought, Action, Action Input, Final Answer 등의 포맷을 명시합니다. 이는 에이전트가 툴 실행과 최종 답변 출력을 구조화된 방식으로 하도록 유도합니다. (예: `Thought: 필요하니 문서검색툴을 써보자 -> Action: DocumentSearchTool -> Action Input: "밀폐 공간 가스 누출 대응"` … 이런 식으로 내부적으로 사고 과정을 거친 뒤 Final Answer 단계에서 결과만 답으로 출력)

에이전트 동작: 에이전트는 **주어진 상황(사용자 질의)**을 받아 다음과 같이 작동합니다:

1. **상황 파악**: 이벤트의 텍스트 설명과 메타데이터를 입력으로 받아 문제를 이해합니다. (예: "작업자 A가 밀폐 공간에서 쓰러짐. 산소 농도 15%로 감소" 등의 설명)
2. **툴 사용 결정**: 에이전트는 내부적으로 "이 질문에 바로 답할 수 있는가? 추가 정보가 필요한가?"를 자문합니다. 대부분의 경우 작업 지시서의 구체 내용(예: 해당 밀폐 공간 작업 지침)을 확인해야 하므로 **DocumentSearchTool**을 호출하는 액션을 선택합니다.
3. **문서 검색**: DocumentSearchTool이 실행되어 벡터DB에서 관련 텍스트 조각들을 반환합니다. 에이전트는 이를 Observation으로 받아 검토합니다. 예를 들어 검색 결과로 "밀폐 공간 작업 지시서 3장: 산소 결핍 시 즉시 환기 및 비상 대피" 등의 내용이 포함될 것입니다.
4. **추론 및 추가 툴 사용**: 에이전트는 검색된 정보를 토대로 현재 위험성을 판단합니다. 필요하면 **RiskAssessmentTool**을 호출하여 "산소 15%면 위험도 높음"이라는 분류를 얻을 수도 있지만, 툴을 별도로 쓰지 않고도 문서 내용과 자체 지식으로 충분히 추론 가능하다면 생략합니다.
5. **최종 답변 생성**: 에이전트 LLM은 최종적으로 사용자에게 전달할 답변을 만듭니다. 이 답변에는 **위험도 수준**, **판단 근거**, **권장 대응조치**가 포함됩니다. 예를 들어:"위험도: 높음입니다. 산소 농도가 15%로 치명적으로 낮아 작업자가 의식을 잃은 것으로 추정됩니다. 즉시 환기를 실시하고 보호 장비를 갖춘 구조팀을 투입하세요. 작업자는 신속히 밀폐 공간에서 구조하여 산소를 공급해야 합니다." 와 같이, 작업지시서에 나온 조치(환기, 구조 등)를 바탕으로 상황에 맞춘 조언을 생성합니다.
6. **출력 포맷팅**: 필요하다면 답변을 구조화된 형태로 출력합니다. (예를 들어 API 응답이라면 JSON 필드에 `risk_level: "높음"`, `actions: ["환기 실시", "구조팀 투입", ...]` 등으로 담고, 사람이 읽는 경우라면 문장 형태로 나열) 에이전트는 프롬프트 지시에 따라 최종 답변만 사용자에게 보여주고, 툴 사용 과정의 내부 Thought/Action 로그는 숨깁니다.

이상의 과정을 통해 에이전트는 사람이 질의하지 않아도 **실시간으로 이벤트를 인지**하고, 관련 문서를 검색하며, **전문지식에 기반한 판단과 조언**을 자동으로 생성합니다. LangChain 에이전트 구조를 활용함으로써, 필요 시 새로운 툴을 추가하거나 프롬프트를 조정하여 기능을 확장하기도 용이합니다 (예: 다른 데이터 소스에 질의하는 툴을 추가하면 에이전트가 상황에 따라 그 툴도 활용 가능).

Hugging Face 모델 로딩 및 추론 구조

SafeGuard AI의 에이전트가 사용하는 **LLM**은 Hugging Face 허브의 오픈소스 모델을 통해 구현합니다. 이를 위해 다음과 같은 모델 구성 전략을 채택합니다:

- **모델 선정**: 산업 안전 도메인 지식이 필요하므로, **정보에 기반한 추론**과 **명령어 수행** 능력이 뛰어난 **Instruction-tuned LLM**을 사용합니다. 예시 후보로는 `Llama-2 13B-chat`, `FLAN-T5 XL`, 또는 한국어가 포함된 다국어 모델(`XLM-RoBERTa` 계열 등) 등이 있습니다. 한국어 작업지시서와 질의를 다룰 경우 한국어 지원이 중요한 만큼, 다국어로 학습된 모델이나 한국어로 퍼인튜닝된 모델을 고르는 것이 좋습니다.

- **모델 로딩**: `models/hf_model.py`에서는 Transformers 라이브러리를 이용해 선택된 모델과 토크나이저를 로드합니다. 예를 들어:

  ```
  from transformers import AutoTokenizer, AutoModelForCausalLM
  tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
  model = AutoModelForCausalLM.from_pretrained(MODEL_NAME, device_map="auto")
  ```

  로 모델을 메모리에 올립니다. `device_map="auto"` 옵션으로 GPU 메모리에 올리고, 필요 시 `torch_dtype=torch.float16` 등으로 메모리 최적화를 합니다. 로딩한 모델은 HuggingFace의 `pipeline` API나 LangChain의 `HuggingFacePipeline`으로 감싸서 사용할 수 있습니다 [6]. 이렇게 하면 LangChain 에이전트가 해당 모델을 LLM으로써 호출하게 됩니다.

- **추론 세부 설정**: 생성시 사용할 파라미터(온도, max_new_tokens 등)를 설정합니다. 예를 들어 정확한 사실 기반 응답을 위해 `temperature=0` (디터미니스틱 출력)를 주거나, 대응 방안 생성 시 약간의 창의성을 부여하려면 0.3 정도로 설정할 수 있습니다. Max token 길이도 작업 지시서 내용 + 응답 길이를 커버할 수 있도록 충분히 크게 잡습니다 (예: 1024 tokens). `HuggingFacePipeline`을 사용할 경우 이러한 매개변수를 `model_kwargs`로 전달하여 모델 출력 특성을 제어합니다 [7].

- **병렬 처리와 응답 시간**: 산업 안전 시스템 특성상 **실시간성**이 중요합니다. 대형 모델 사용 시 추론 속도가 느릴 수 있으므로, 개선 방안을 고려합니다. 하나는 HuggingFace의 텍스트 생성 전용 서버(Transformers Serving 또는 Text Generation Inference)를 사용해 모델 추론을 최적화하는 것이고, 다른 하나는 모델 크기를 줄이는 것입니다. 예컨대 7억~130억 파라미터 규모 모델을 사용하되 4-bit 양자화 등을 적용해 추론을 빠르게 하는 방법이 있습니다. 또한 FastAPI 서버에서 이벤트당 에이전트 호출을 비동기로 처리하거나, 멀티스레드로 동시 이벤트를 다루도록 하여 응답 지연을 최소화합니다.

- **모델 출력 후처리**: 모델이 생성한 최종 답변 텍스트는 필요에 따라 후처리합니다. 예를 들어 Bullet point 형태로 여러 조치가 나왔다면 보기 좋게 포맷하거나, 전문용어를 현장 작업자가 이해할 수 있는 용어로 변환하는 등의 작업입니다. 이러한 후처리는 LLM에 바로 시킬 수도 있고, 파이프라인 마지막에 별도 함수로 수행할 수도 있습니다.

정리하면, Hugging Face Transformers 모델을 **한번 로딩하여 재사용**함으로써 추론마다 일어나는 부하를 줄이고, LangChain 통합을 통해 에이전트가 자연스럽게 모델을 활용하도록 구성합니다. 오픈소스 모델의 이점은 **내부 동작을 통제하고 사용자 데이터 프라이버시를 지킬 수 있다는 것**이며, 필요시 도메인 데이터로 추가 파인튜닝하여 성능을 높이는 것도 가능합니다. (예: 과거 사고 보고서 데이터를 활용한 지속 학습)

사용 시나리오 흐름

SafeGuard AI의 동작을 **이상 상황 발생** 시나리오를 통해 단계별로 설명하면 다음과 같습니다:

1. **이벤트 발생 및 수집**: 작업 현장에서 스마트 카메라 또는 센서가 이상행동/조건을 감지합니다. 예를 들어 "작업자 보호구 미착용"이나 "지게차 충돌", "온도 급상승" 같은 이벤트가 트리거됩니다. 이때 해당 **이미지**(프레임)와 관련 **메타데이터**(이벤트 종류, 발생 시간, 위치, 센서 수치 등)가 SafeGuard AI 시스템으로 전송됩니다. FastAPI 엔드포인트로 이벤트 정보 JSON이 들어오거나, 메시지 큐 등을 통해 이벤트가 전달될 수 있습니다.

2. **초기 상황 해석**: 전달된 이미지와 메타데이터를 바탕으로 우선 간략한 **상황 설명 문자열**이 만들어집니다. (이미지 자체는 현재 RAG 파이프라인에 직접적으로 넣지 않으므로, 사전에 CV 모듈이 "작업자가 안전모를 착용하지 않음" 같은 텍스트 태그를 붙여줄 수 있습니다.) 예를 들어 이벤트 메타데이터를 조합한 설명: "16시 20분, 3공장 라인에서 작업자 A가 안전모 미착용 상태로 작업 중인 것이 검출됨." 이러한 텍스트는 이후 에이전트의 입력으로 활용됩니다.

3. **관련 작업지시서 검색**: 에이전트가 상황 설명을 받아 대응 방안을 찾는 **문제 해결 단계**를 시작합니다. 우선 해당 상황과 연관된 안전 수칙이나 절차를 찾기 위해 **DocumentSearchTool** 툴을 호출합니다. 내부적으로 벡터DB Retriever가 "안전모 미착용", "개인 보호구 관련 지침" 등의 키워드로 임베딩된 문서 조각을 유사도 검색합니다. 그 결과, 예컨대 "개인보호구 착용 지침 PDF"의 relevant section이 반환됩니다. 내용에는 "모든 작업자는 작업장 진입 시 안전모 등 보호구를 착용해야 하며, 위반 시 작업 중지 및 교육 실시" 등의 문구가 포함되어 있을 것입니다. 에이전트는 이 검색 결과 텍스트를 입력 받아 참고합니다.

4. **상황에 대한 판단 (위험 평가)**: 에이전트는 검색된 지침 내용과 현재 상황을照合하여 위험도를 평가합니다. 이 사례에서 안전모 미착용 자체는 즉각적인 사고는 아니지만 **잠재적 위험**으로 분류됩니다. 지침서에는 위반 시 제재 조치 등이 언급되어 있으므로, 에이전트는 이를 근거로 위험도를 "중간" 정도로 판단할 수 있습니다. (필요하다면 RiskAssessmentTool이 미리 정의된 분류를 줄 수도 있으나, 여기서는 LLM이 스스로 문맥을 이해해 결정한다고 가정합니다.)

5. **대응 방안 도출**: 에이전트는 **최종 답변 생성 단계**로서, 해당 상황에 취할 권장 조치를 작성합니다. 작업지시서에서 찾은 내용에 따르면 안전모 미착용 시 즉시 시정 조치와 재교육이 필요하므로, 에이전트 답변에 "작업을 일시 중지시키고 해당 작업자에게 보호구를 지급 및 착용토록 하십시오. 추후 해당 작업자에 대한 안전 교육을 실시하여 재발을 방지해야 합니다."라는 제안을 포함시킵니다. 추가로, 현장 관리자에게 알림을 보내는 등의 일반적인 조치도 언급할 수 있습니다. 답변에는 **위험도 등급** (중간)과 **권장 대응조치**들이 명시됩니다.

6. **응답 전송 및 후속 조치**: 생성된 대응 방안은 API 응답으로 프론트엔드 대시보드나 관리자 모바일앱 등에 전송됩니다. 관리자는 이를 확인하고 필요시 조치를 실행에 옮깁니다. SafeGuard AI는 응답 내용을 **템플릿화**하여 리포트도 자동으로 작성할 수 있습니다 (예: "Incident Report: 위험도-중간, 원인-보호구 미착용, 조치-OO" 양식). 또한 시스템은 이 이벤트와 에이전트의 조치 제안을 데이터베이스에 로그로 남겨, 향후 유사 이벤트 발생 시 참고하거나 시스템 성능을 개선하는데 활용합니다.

이와 같은 흐름에서 주목할 점은, **에이전트가 사람이 개입하지 않아도 현장의 맥락을 이해하고 적절한 지식**을 찾아내어 **조치 제안을 자동으로 생성**한다는 것입니다. 즉, RAG 에이전트는 사전에 학습된 LLM 지식과 사내 문서로부터 검색한 최신 지식을 결합하여, 상황별 맞춤형 답변을 만들어냅니다. 이를 통해 현장 대응 속도를 높이고, 작업자들에게 일관되고 정확한 안전 지침을 실시간으로 제공할 수 있습니다.

기술 스택

SafeGuard AI 에이전트형 RAG 구현에는 다음과 같은 기술이 사용됩니다:

- **Python**: 전체 시스템 구현 언어. 데이터 처리, 모델 추론, API 서버 등 백엔드 로직을 파이썬으로 작성합니다.

- **LangChain**: LLM 오케스트레이션 프레임워크. 문서 로더, 텍스트 분할, 임베딩, 벡터스토어 연동, 에이전트/툴 체인 등 RAG 구성 요소를 연결하는 고급 추상화를 제공합니다. LangChain을 활용하여 Retriever, Tool, Agent를 손쉽게 구성하고, 프롬프트 관리 및 디버깅(Trace)도 체계화했습니다.
- **Hugging Face Transformers**: 오픈소스 **LLM 및 임베딩 모델** 라이브러리. Hugging Face 허브의 사전 학습 언어모델(예: T5, Llama 계열)을 로컬에 로드하여 사용합니다. 또한 `sentence-transformers` 모델 등 임베딩 생성을 위해 Transformers 기반 모델을 활용합니다. 이 스택을 통해 외부 API 없이 **온프레미스에서 추론**이 가능하며, 필요하면 추가 파인튜닝도 동일한 생태계에서 수행할 수 있습니다.
- **벡터 데이터베이스**: **FAISS** 또는 **OpenSearch**를 사용하여 임베딩 벡터를 저장하고 유사도 검색을 지원합니다.
- FAISS는 C++ 기반 라이브러리를 Python 바인딩으로 사용하며, milvus, Chroma 등과 함께 간편한 벡터스토어로 널리 활용됩니다.
- OpenSearch는 AWS에서 관리형 서비스로도 제공되며, 대규모 데이터에 적합합니다 [3]. 두 솔루션 모두 LangChain에 커넥터가 있어 선택 가능하며, 현재 구현은 개발 편의상 FAISS로 시작하고 확장성 요구에 따라 OpenSearch로 이행할 수 있도록 추상화해두었습니다.
- **PyMuPDF / pdfminer.six**: PDF **문서 로딩/파싱** 라이브러리. PyMuPDF는 PDF 페이지를 하나씩 읽어 텍스트와 글자 좌표 등을 추출할 수 있고 성능이 뛰어나, 작업지시서와 같은 스캐너 PDF도 OCR 없이 텍스트 레이어를 읽어낼 수 있습니다. pdfminer.six도 텍스트 추출에 쓰일 수 있으나, 속도가 느려서 주로 PyMuPDF(Fitz)를 채택했습니다.
- **FastAPI** (선택): 에이전트 기능을 실시간 서비스로 제공하기 위한 **웹 프레임워크**입니다. REST API 형태로 `/analyze_event` 등의 엔드포인트를 만들고, 외부 시스템(예: 이벤트 발생을 감지한 IoT 플랫폼)이 해당 API를 호출하면 에이전트의 분석 결과를 JSON으로 반환합니다. FastAPI는 비동기 요청 처리를 지원하여 다수의 동시 이벤트도 효율적으로 처리할 수 있고, Swagger UI로 테스트도 용이해 선택되었습니다.
- **기타**: 이밖에 NumPy, Pandas 등 데이터 처리 라이브러리와, Pydantic (데이터 검증), Uvicorn (ASGI 서버) 등이 보조적으로 사용됩니다. 로그 수집을 위해 Loguru, 모니터링을 위해 Prometheus 연동 등을 고려하고 있습니다. 프론트엔드는 별도로 React 등을 활용해 대시보드를 구성하며, 여기서는 백엔드 RAG 시스템 설계에 중점을 둡니다.

확장 고려사항

제안된 설계는 기본적인 RAG 에이전트 시스템의 뼈대를 제공합니다. 향후 **기능 확장**이나 **운영 상의 개선**을 위해 다음 사항을 추가로 고려합니다:

- **다국어 작업지시서 대응**: 글로벌 사업장을 가진 환경이라면 작업지시서가 영어, 한국어 등 혼용될 수 있습니다. 이를 다루기 위해 **다국어 임베딩**과 **다국어 LLM**을 지원해야 합니다. 벡터DB 색인 시 각 문서의 언어 정보를 메타데이터로 저장하고, 질의어와 문서 언어가 불일치하면 **교차 언어 임베딩 모델**을 사용합니다. 예를 들어 Cohere의 multilingual embedding이나 LaBSE 등을 활용하면 한글 질의로 영어 문서를 검색할 수 있습니다. 또는 **번역 단계**를 삽입해 질의를 벡터화 전에 영어로 번역하고, 검색된 영문 문서를 다시 한글로 번역하는 파이프라인을 고려할 수 있습니다. LLM 또한 지원 언어를 고려해 선택해야 하며, 필요하면 질의나 답변을 번역하는 툴을 에이전트에 추가할 수 있습니다. 이렇게 하면 시스템이 언어 장벽 없이 동일한 기능을 수행할 수 있습니다.

- **상황 대응 템플릿 자동 생성**: 새로운 유형의 위험 상황에 대해 **표준 대응 절차**를 자동으로 생성/갱신하는 기능을 확장할 수 있습니다. 예를 들어 과거에 없던 유형의 사고가 발생하면, 해당 사례를 바탕으로 LLM에게 **임시 작업 지침서** 초안을 작성하게 한 뒤 도메인 전문가가 검토하여 공식 지침으로 채택하는 프로세스입니다. 이를 위해 에이전트에 "템플릿 생성" 모드를 두어, 특정 입력에 대해서는 답변 대신 체크리스트 형식의 문서를 만들도록 프롬프트를 달리할 수 있습니다. 또한 대응 방안 출력 자체를 템플릿화하여 **정형 리포트**를 자동 작성하게 할 수 있습니다. 예컨데, 위험 평가 리포트 양식을 미리 정의해 두고 LLM이 `{{사고유형}}, {{원인}}, {{조치}}` 같은 자리에 들어갈 내용을 채워넣는 식입니다. LangChain의 PromptTemplate 기능을 활용하면 이러한 **포맷 지정 응답**을 얻을 수 있습니다. 이 기능은 향후 사고 데이터베이스를 축적하고, 유사 사건에 대한 대응 일관성을 높이는 데 기여합니다.

- **모델 및 데이터 확장**: 시간이 지남에 따라 작업 지시서가 추가/변경되거나 LLM의 성능을 높여야 할 필요가 생깁니다. 벡터DB에 **새 문서 추가**는 ingestion 파이프라인을 주기적으로 또는 이벤트 기반으로 돌려주면 되고, 변경된 지침서는 임베딩을 업데이트하여 인덱스를 리프레시합니다. LLM의 경우 더 성능 좋은 최신 모델로 교체하거나, 현장 데이터로 퍼인튜닝(Fine-tuning)하여 전문용어 이해도와 응답 정확도를 향상시킬 수 있습니다. 설계 구조상 `models/hf_model.py`나 설정만 변경하면 모델 교체가 가능하므로 업그레이드가 비교적 수월합니다. 단, 모델이 변경되면 응답 양식이나 프롬프트 튜닝을 재검토하여, 이전 대비 변화된 응답 경향에 맞게 수정을 가해야 합니다.

- **안전성과 모니터링**: 산업 안전 도메인의 특성상 잘못된 답변은 큰 위험을 초래할 수 있습니다. 따라서 에이전트의 출력을 **검증**하고, 필요시 사람의 승인 후 실행하는 **Human-in-the-loop** 절차를 유지합니다. 확장으로, LLM의 출력을 검증하는 별도 **판단 모델**(예: 응답이 지침서 사실과 일치하는지 판정하는 LLM-as-a-judge)을 도입할 수 있습니다. 또한 시스템 사용 로그를 모니터링하여, 에이전트가 어떤 질문에 어떤 근거로 답했는지 **투명하게 기록**하고 나중에 감사할 수 있게 합니다. 이러한 장치는 향후 시스템 신뢰성을 높이고 규제를 준수하는 데 도움이 됩니다.

요약하면, SafeGuard AI의 RAG 에이전트는 **모듈화된 설계**를 바탕으로 초기 요구사항을 충족시키면서, 다국어 지원이나 템플릿 자동화 등 향후 기능 추가에도 유연하게 대응할 수 있습니다. 이 설계를 토대로 구현을 진행하면, 실시간 위험 분석 및 대응 제안이라는 목표에 한 걸음 더 다가갈 수 있을 것입니다.

---

1 What's the Best Python Library for Extracting Text from PDFs? : r/LangChain
https://www.reddit.com/r/LangChain/comments/1e7cntq/whats_the_best_python_library_for_extracting_text/

2 Designing RAG pipelines using LangChain and Evaluating them using Ragas (v0.1.7) | by Rhitesh Kumar Singh | Medium
https://medium.com/@rhitesh.ksingh99/designing-rag-pipelines-using-langchain-and-evaluating-them-using-ragas-4e7d09262bac

3 4 Using Langchain and OpenSearch for RAG on AWS | Caylent
https://caylent.com/blog/building-a-rag-with-lang-chain-on-aws

5 6 7 Implementing RAG with Langchain and Hugging Face | by Akriti Upadhyay | Medium
https://medium.com/@akriti.upadhyay/implementing-rag-with-langchain-and-hugging-face-28e3ea66c5f7