통합 프로젝트 명세서 - 정부 공문서 AI 검색 서비스

1. 프로젝트 개요

1.1 프로젝트명

일반시민을 위한 정부 공문서 AI 검색 서비스

1.2 프로젝트 목적

- 행정안전부에서 제공하는 정부 공문서 AI 학습데이터를 활용한 시민 친화적 정보 검색 시스템 구축
- 복잡한 정부 정책과 공문서를 일반 시민이 쉽게 이해할 수 있도록 돕는 AI Agent 서비스 개발
- RAG 기술을 활용하여 정확하고 이해하기 쉬운 정보 제공

1.3 시스템 아키텍처 개요

```
[클라이언트 (Streamlit)] ↔ [서버 (FastAPI)] ↔ [Vector DB (FAISS) + Cache (메모리)]
                                ↓
                        [Azure OpenAI API + LangGraph]
```

2. 기술 스택 (통일)

2.1 클라이언트 (Frontend)

- **Framework**: Streamlit
- **HTTP Client**: httpx
- **상태 관리**: Streamlit session state
- **UI**: Streamlit 기본 컴포넌트 + 커스텀 CSS

2.2 서버 (Backend)

- **Framework**: FastAPI
- **AI Framework**: LangChain + LangGraph
- **Vector Database**: **FAISS** (메인)
- **Embedding**: Azure OpenAI text-embedding-3-large
- **LLM**: Azure OpenAI GPT-4o-mini/GPT-4o
- **캐시**: **메모리 기반 캐시** (Redis 제외)
- **세션 관리**: **SQLite**
- **HTTP Client**: httpx

2.3 개발 및 배포

- **컨테이너화**: Docker + Docker Compose
- **테스트**: pytest + httpx
- **로깅**: Python logging (JSON 포맷)
- **모니터링**: **기본 로깅** (Prometheus 제외)

3. Multi-Agent 시스템 설계 (확정)

3.1 Agent 구성

```
python

1. CitizenQueryAnalyzer Agent
   - 시민의 일상 언어 질의를 분석
   - 의도 파악 및 키워드 추출
   - 관련 카테고리 분류

2. PolicyDocumentRetriever Agent
   - FAISS Vector DB에서 관련 문서 검색
   - 문서 관련성 평가 및 랭킹
   - 다중 소스 정보 통합

3. CitizenFriendlyProcessor Agent
   - 공문서를 시민 친화적 언어로 변환
   - 복잡한 절차를 단계별로 정리
   - 핵심 정보 추출 및 구조화

4. InteractiveResponseGenerator Agent
   - 최종 답변 생성
   - 추가 질문 제안
   - 관련 정보 링크 제공
```

3.2 LangGraph 워크플로우

```
사용자 질의 → 질의 분석 → 문서 검색 → 내용 처리 → 응답 생성
   ↓         ↓         ↓         ↓         ↓
  세션 관리 → 카테고리 분류 → 관련성 평가 → 용어 변환 → 대화형 응답
```

4. 시스템 아키텍처 (확정)

4.1 프로젝트 구조

```
project_root/
├── streamlit_app/              # 클라이언트
```

```
│   ├── main.py
│   ├── pages/
│   ├── components/
│   ├── services/
│   └── utils/
├── fastapi_server/          # 서버
│   ├── main.py
│   ├── api/
│   ├── core/
│   │   ├── agents/
│   │   ├── services/
│   │   └── workflow/
│   ├── models/
│   └── utils/
├── data/                    # 데이터
│   ├── documents/
│   ├── vector_db/
│   └── sessions.db
├── tests/                   # 테스트
├── requirements.txt
├── requirements-client.txt
├── docker-compose.yml
└── .env.example
```

4.2 데이터베이스 설계

python

 # Vector Database (FAISS)
- 문서 임베딩: 768차원 벡터
- 메타데이터: 제목, 카테고리, 발행일, 난이도 등
- 인덱스: IVF (고속 검색)

 # 세션 DB (SQLite)
- ChatSession: 세션 정보
- ChatMessage: 대화 기록
- SearchHistory: 검색 기록

5. API 설계 (확정)

5.1 주요 엔드포인트

python

```
# 검색 API
POST /api/v1/search/query
GET /api/v1/search/categories
GET /api/v1/search/popular

# 대화형 API
POST /api/v1/chat/message
GET /api/v1/chat/history/{session_id}
POST /api/v1/chat/session

# 관리 API
GET /api/v1/health
GET /api/v1/metrics
```

5.2 데이터 모델

```python
class SearchRequest(BaseModel):
    query: str
    category: Optional[str] = None
    max_results: int = 5
    session_id: Optional[str] = None

class SearchResponse(BaseModel):
    results: List[DocumentResult]
    summary: str
    total_count: int
    processing_time: float
    suggestions: List[str]
    confidence_score: float
```

6. 개발 일정 (7일 확정)

Phase 1: 설계 및 환경 구축 (Day 1)

오전 (4시간)

- [ ] 프로젝트 구조 생성
- [ ] 환경 설정 (requirements.txt, .env, config)
- [ ] 기본 테스트 프레임워크 설정
- [ ] Docker 환경 구성

오후 (4시간)

- [ ] LangGraph 상태 시스템 설계

- [ ] Agent 인터페이스 정의
- [ ] API 스키마 정의 (FastAPI)
- [ ] Streamlit 기본 구조 생성

Phase 2: 핵심 Agent 구현 (Day 2-3)

Day 2: Multi-Agent 시스템

- [ ] CitizenQueryAnalyzer 구현 + 테스트
- [ ] PolicyDocumentRetriever 구현 + 테스트
- [ ] CitizenFriendlyProcessor 구현 + 테스트
- [ ] InteractiveResponseGenerator 구현 + 테스트

Day 3: 워크플로우 통합

- [ ] LangGraph 워크플로우 구성
- [ ] Agent 간 상태 전달 구현
- [ ] 조건부 라우팅 및 에러 처리
- [ ] 통합 테스트

Phase 3: API 및 UI 개발 (Day 4-5)

Day 4: FastAPI 백엔드

- [ ] FastAPI 앱 구조 완성
- [ ] 모든 API 엔드포인트 구현
- [ ] 미들웨어 및 보안 설정
- [ ] API 테스트 작성

Day 5: Streamlit 프론트엔드

- [ ] 전체 UI 페이지 구현
- [ ] API 클라이언트 연동
- [ ] 시민 친화적 UX 최적화
- [ ] 프론트엔드 테스트

Phase 4: 데이터 구축 (Day 5-6)

Day 5 오후: 데이터 처리

- [ ] 정부 공문서 데이터 전처리
- [ ] FAISS Vector DB 구축
- [ ] 임베딩 생성 및 인덱싱
- [ ] 검색 성능 최적화

Day 6: 통합 및 최적화

- [ ] 전체 시스템 E2E 테스트
- [ ] 성능 최적화 (응답시간 < 3초)
- [ ] 메모리 사용량 최적화
- [ ] Docker 컨테이너화

Phase 5: 배포 및 문서화 (Day 7)

오전 (4시간)

- [ ] Docker Compose 배포 환경 완성
- [ ] 기본 로깅 시스템 구현
- [ ] 최종 테스트 및 버그 수정

오후 (4시간)

- [ ] API 문서 자동 생성
- [ ] 사용자 가이드 작성
- [ ] README 및 개발 문서 완성
- [ ] 프로젝트 제출 준비

7. 성능 및 품질 요구사항 (확정)

7.1 성능 지표

- **검색 응답 시간**: 평균 3초 이내
- **대화 응답 시간**: 평균 5초 이내
- **정확도**: 관련 문서 검색 정확도 80% 이상
- **동시 사용자**: 최소 20명 동시 접속 지원

7.2 기술적 제약사항

- **메모리 사용량**: 8GB 이하
- **디스크 사용량**: Vector DB 포함 10GB 이하
- **API Rate Limit**: 분당 30회
- **세션 유지**: 2시간

8. 개발 환경 설정 (확정)

8.1 필수 환경 변수

```bash
```

```
# Azure OpenAI
AOAI_ENDPOINT=https://skcc-atl-dev-openai-01.openai.azure.com/
AOAI_API_KEY=5kWBVPecvaqFeiB3PJYXnfkMHcIg66duBhVXY5IWE9z187WonAnAJQQJ99BBACYeBjFXJ3w3AAABACOG
AOAI_DEPLOY_GPT4O_MINI=gpt-4o-mini
AOAI_DEPLOY_GPT4O=gpt-4o
AOAI_DEPLOY_EMBED_3_LARGE=text-embedding-3-large

# 애플리케이션
CLIENT_URL=http://localhost:8501
SERVER_URL=http://localhost:8000
VECTOR_DB_PATH=./data/vector_db
SESSION_DB_PATH=./data/sessions.db
LOG_LEVEL=INFO
```

8.2 Docker 배포

yaml

```yaml
# docker-compose.yml (간소화)
version: '3.8'
services:
  api-server:
    build:
      context: .
      dockerfile: Dockerfile.server
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
    environment:
      - AOAI_ENDPOINT=${AOAI_ENDPOINT}
      - AOAI_API_KEY=${AOAI_API_KEY}

  web-client:
    build:
      context: .
      dockerfile: Dockerfile.client
    ports:
      - "8501:8501"
    depends_on:
      - api-server
    environment:
      - SERVER_URL=http://api-server:8000
```

9. 테스트 전략 (확정)

9.1 테스트 유형

- **단위 테스트**: 각 Agent 개별 기능
- **통합 테스트**: Agent 간 워크플로우
- **API 테스트**: FastAPI 엔드포인트
- **E2E 테스트**: 전체 사용자 시나리오

9.2 테스트 커버리지 목표

- **코드 커버리지**: 85% 이상
- **API 커버리지**: 95% 이상
- **시나리오 커버리지**: 주요 사용자 시나리오 100%

10. 클로드 AI 협업 최적화 전략

10.1 효율적인 요청 패턴

1. 배치 처리: "Agent A, B, C를 TDD로 모두 구현해줘"
2. 컨텍스트 유지: 이전 설계 결정사항 포함
3. 점진적 개선: 기본 → 고급 기능 순서
4. 통합 개발: 테스트 + 구현 + 문서 한번에

10.2 시간 분배 (7일)

- **클로드 고속 개발**: 5일 (70%)
- **수동 실행/디버깅**: 1.5일 (25%)
- **최종 검증/문서화**: 0.5일 (5%)

11. 위험 요소 및 대응 방안

11.1 기술적 위험

- **Azure OpenAI API 제한**: Rate limiting 및 재시도 로직
- **Vector 검색 성능**: FAISS 인덱스 최적화
- **메모리 사용량**: 배치 크기 조정 및 캐시 관리

11.2 일정 위험

- **복잡성 증가**: MVP 우선, 고급 기능 후순위
- **통합 이슈**: 일일 통합 테스트로 조기 발견
- **성능 이슈**: 병목 지점 사전 식별 및 최적화

12. 성공 기준 (최종)

12.1 기능적 완성도

- [ ] 자연어 질의 처리 (95%)
- [ ] 시민 친화적 답변 생성 (90%)
- [ ] 대화형 상담 기능 (85%)
- [ ] 카테고리별 검색 (95%)

12.2 기술적 완성도

- [ ] Multi-Agent 시스템 구현 (100%)
- [ ] RAG 파이프라인 구현 (100%)
- [ ] API 완전 구현 (95%)
- [ ] Docker 배포 준비 (95%)

12.3 사용자 경험

- [ ] 직관적인 UI/UX (90%)
- [ ] 빠른 응답 시간 (< 3초)
- [ ] 정확한 검색 결과 (80% 이상)
- [ ] 안정적인 서비스 운영

이 통합 명세서를 따라 개발하면 **7일 내에 고품질의 정부 공문서 AI 검색 서비스를 완성**할 수 있습니다.