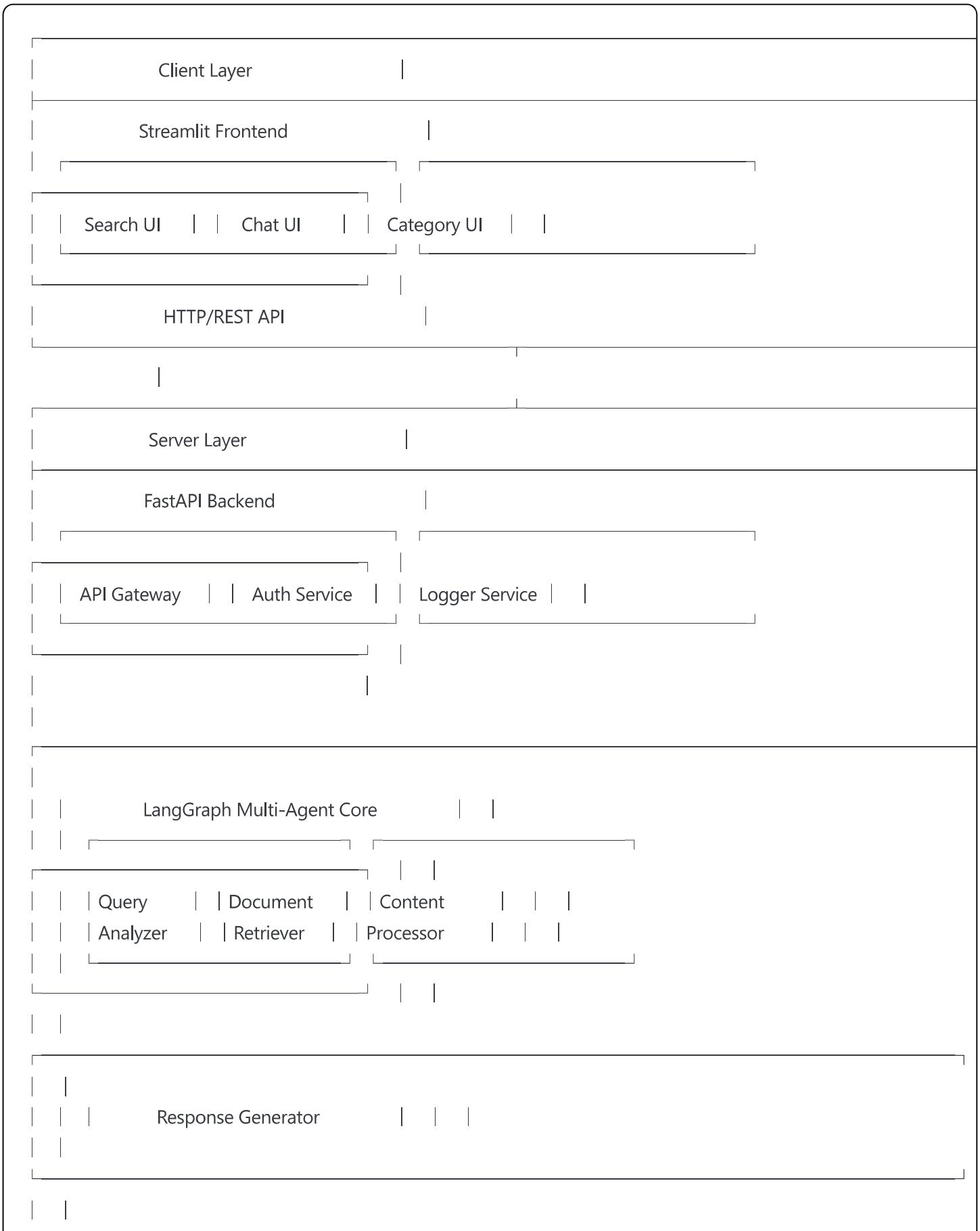
일반시민을 위한 정부 공문서 AI 검색 서비스 - 시스템 설계문서

1. 시스템 아키텍처 설계

1.1 전체 시스템 아키텍처



1.2 컴포넌트별 설계

1.2.1 클라이언트 레이어 (Streamlit)

```python
```

```
# 프로젝트 구조
streamlit_app/
├── main.py                 # 메인 앱 진입점
├── config/
│   ├── __init__.py
│   ├── settings.py         # 클라이언트 설정
│   └── constants.py        # 상수 정의
├── pages/
│   ├── __init__.py
│   ├── home.py             # 홈페이지
│   ├── search.py           # 검색 페이지
│   ├── chat.py             # 대화형 상담
│   ├── categories.py       # 카테고리 탐색
│   └── history.py          # 검색 기록
├── components/
│   ├── __init__.py
│   ├── search_box.py       # 검색 입력 컴포넌트
│   ├── result_card.py      # 결과 카드 컴포넌트
│   ├── chat_bubble.py      # 채팅 말풍선
│   ├── category_grid.py    # 카테고리 그리드
│   └── loading_spinner.py  # 로딩 스피너
├── services/
│   ├── __init__.py
│   ├── api_client.py       # API 클라이언트
│   └── session_manager.py  # 세션 관리
├── utils/
│   ├── __init__.py
│   ├── ui_helpers.py       # UI 유틸리티
│   ├── validators.py       # 입력 검증
│   └── formatters.py       # 데이터 포맷팅
└── assets/
    ├── styles.css          # 커스텀 CSS
    └── images/             # 이미지 리소스
```

1.2.2 서버 레이어 (FastAPI)

python

```
# 프로젝트 구조
fastapi_server/
├── main.py                 # FastAPI 앱 진입점
├── config/
│   ├── __init__.py
│   ├── settings.py         # 서버 설정
│   ├── database.py         # 데이터베이스 설정
│   └── logging.py          # 로깅 설정
├── api/
│   ├── __init__.py
│   ├── dependencies.py     # 의존성 주입
│   └── v1/
│       ├── __init__.py
│       ├── search.py       # 검색 API
│       ├── chat.py         # 채팅 API
│       ├── categories.py   # 카테고리 API
│       └── health.py       # 헬스 체크
├── core/
│   ├── __init__.py
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── base_agent.py           # 기본 Agent 클래스
│   │   ├── citizen_query_analyzer.py    # 시민 질의 분석
│   │   ├── policy_document_retriever.py # 정책 문서 검색
│   │   ├── citizen_friendly_processor.py # 시민 친화적 처리
│   │   └── interactive_response_generator.py # 응답 생성
│   ├── services/
│   │   ├── __init__.py
│   │   ├── rag_service.py          # RAG 파이프라인
│   │   ├── vector_service.py       # Vector DB 서비스
│   │   ├── openai_service.py       # Azure OpenAI 서비스
│   │   ├── cache_service.py        # 캐시 서비스
│   │   └── session_service.py      # 세션 서비스
│   └── workflow/
│       ├── __init__.py
│       ├── search_workflow.py      # 검색 워크플로우
│       └── chat_workflow.py        # 채팅 워크플로우
├── models/
│   ├── __init__.py
│   ├── requests/           # 요청 모델
│   │   ├── search_models.py
│   │   └── chat_models.py
│   ├── responses/          # 응답 모델
│   │   ├── search_models.py
│   │   └── chat_models.py
│   └── entities/           # 엔터티 모델
```

```
│       ├── document.py
│       └── session.py
├── utils/
│   ├── __init__.py
│   ├── exceptions.py        # 커스텀 예외
│   ├── validators.py        # 검증 유틸리티
│   └── helpers.py           # 헬퍼 함수
└── data/
    ├── __init__.py
    ├── preprocessor.py      # 데이터 전처리
    ├── vector_builder.py    # Vector DB 구축
    └── term_dictionary.py   # 용어 사전
```

2. Multi-Agent 시스템 설계

2.1 Agent 아키텍처

python

```python
# LangGraph 기반 Multi-Agent 워크플로우
from langgraph.graph import StateGraph, END
from typing import TypedDict, List, Optional

class AgentState(TypedDict):
    """Agent 간 공유되는 상태"""
    user_query: str
    processed_query: dict
    retrieved_documents: List[dict]
    processed_content: dict
    final_response: dict
    session_id: str
    user_context: dict
    error_message: Optional[str]
```

2.2 개별 Agent 설계

2.2.1 CitizenQueryAnalyzer Agent

python

```python
class CitizenQueryAnalyzer:
    """시민 질의 분석 Agent"""

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

    async def analyze_query(self, state: AgentState) -> AgentState:
        """
        사용자 질의를 분석하여 의도와 키워드를 추출

        수행 작업:
        1. 자연어 질의를 구조화된 쿼리로 변환
        2. 생활 용어를 공문서 용어로 매핑
        3. 질의 의도 분류 (정보조회, 신청방법, 자격요건 등)
        4. 관련 카테고리 및 키워드 추출
        """

        prompt = f"""
        당신은 일반 시민의 질문을 분석하는 전문가입니다.

        사용자 질문: {state['user_query']}

        다음 작업을 수행해주세요:
        1. 질문의 핵심 의도를 파악하세요 (정보조회/신청방법/자격요건/기타)
        2. 생활 용어를 공문서 용어로 변환하세요
        3. 관련 정책 카테고리를 식별하세요 (복지/세금/교육/주택/취업 등)
        4. 검색에 필요한 핵심 키워드를 추출하세요

        응답 형식:
        {{
            "intent": "질문 의도",
            "category": "관련 카테고리",
            "keywords": ["키워드1", "키워드2"],
            "formal_terms": ["공문서용어1", "공문서용어2"],
            "query_type": "검색 유형"
        }}
        """

        # LLM 호출 및 결과 파싱
        result = await self.llm.apredict(prompt)

        state['processed_query'] = self.parse_analysis_result(result)
        return state
```

2.2.2 PolicyDocumentRetriever Agent

python

```python
class PolicyDocumentRetriever:
    """정책 문서 검색 Agent"""

    def __init__(self, vector_service, embedding_service):
        self.vector_service = vector_service
        self.embedding_service = embedding_service

    async def retrieve_documents(self, state: AgentState) -> AgentState:
        """
        Vector DB에서 관련 문서를 검색

        수행 작업:
        1. 쿼리 임베딩 생성
        2. 유사도 기반 문서 검색
        3. 카테고리 필터링 적용
        4. 문서 관련성 평가 및 랭킹
        5. 메타데이터 기반 후처리
        """

        processed_query = state['processed_query']

        # 1. 검색 쿼리 임베딩
        search_embedding = await self.embedding_service.embed_query(
            processed_query['keywords'] + processed_query['formal_terms']
        )

        # 2. Vector DB 검색
        raw_results = await self.vector_service.similarity_search(
            embedding=search_embedding,
            k=20,  # 더 많이 검색 후 필터링
            filter={
                "category": processed_query['category'],
                "relevance_score": {"$gte": 0.7}
            }
        )

        # 3. 문서 관련성 재평가
        ranked_documents = await self.rank_documents(
            raw_results,
            processed_query
        )

        state['retrieved_documents'] = ranked_documents[:5]
        return state

    async def rank_documents(self, documents, query_info):
```

```
    """문서 관련성 기반 재랭킹"""
    # 생활 연관성, 최신성, 완전성 등을 종합하여 점수 계산
    pass
```

2.2.3 CitizenFriendlyProcessor Agent

```
python
```

```python
class CitizenFriendlyProcessor:
    """시민 친화적 콘텐츠 처리 Agent"""

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

    async def process_content(self, state: AgentState) -> AgentState:
        """
        검색된 문서를 시민 친화적으로 가공

        수행 작업:
        1. 공문서 용어를 일반 용어로 변환
        2. 복잡한 절차를 단계별로 정리
        3. 핵심 정보 추출 (신청방법, 준비서류, 기한 등)
        4. FAQ 형태로 구조화
        """

        documents = state['retrieved_documents']
        query_intent = state['processed_query']['intent']

        processed_content = {
            "summary": "",
            "key_info": {},
            "step_by_step": [],
            "requirements": [],
            "deadlines": [],
            "contact_info": {},
            "related_policies": []
        }

        # 의도별 맞춤 처리
        if query_intent == "신청방법":
            processed_content = await self.process_application_info(documents)
        elif query_intent == "자격요건":
            processed_content = await self.process_eligibility_info(documents)
        elif query_intent == "정보조회":
            processed_content = await self.process_general_info(documents)

        state['processed_content'] = processed_content
        return state

    async def process_application_info(self, documents):
        """신청 방법 정보 처리"""
        prompt = f"""
        다음 공문서들을 바탕으로 일반 시민이 이해하기 쉽도록 신청 방법을 정리해주세요.
```

```
문서 내용: {documents}

다음 형식으로 정리해주세요:
1. 한 줄 요약 (20자 이내)
2. 신청 자격 (간단명료하게)
3. 준비 서류 (목록 형태)
4. 신청 절차 (단계별)
5. 신청 기한
6. 문의처

전문 용어는 괄호 안에 쉬운 설명을 추가해주세요.
"""

# LLM 처리 및 구조화
pass
```

2.2.4 InteractiveResponseGenerator Agent

python

```
class InteractiveResponseGenerator:
    """대화형 응답 생성 Agent"""

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

    async def generate_response(self, state: AgentState) -> AgentState:
        """
        최종 사용자 응답 생성

        수행 작업:
        1. 처리된 콘텐츠를 자연스러운 대화형 응답으로 변환
        2. 추가 질문 제안 생성
        3. 관련 정보 링크 제공
        4. 사용자 맞춤 조언 추가
        """

        processed_content = state['processed_content']
        user_context = state.get('user_context', {})

        response = {
            "main_answer": "",
            "quick_summary": "",
            "detailed_info": {},
            "next_questions": [],
            "helpful_tips": [],
            "source_links": [],
            "confidence_score": 0.0
        }

        # 대화형 응답 생성
        prompt = f"""
        사용자의 질문에 대한 친근하고 도움이 되는 답변을 생성해주세요.

        사용자 질문: {state['user_query']}
        처리된 정보: {processed_content}

        다음 원칙을 지켜주세요:
        1. 친근하고 이해하기 쉬운 톤앤매너
        2. 핵심 정보를 앞에 배치
        3. 구체적인 행동 방법 제시
        4. 추가로 궁금할 만한 내용 제안
        5. 어려운 용어 최소화

        응답 구조:
        - 핵심 답변 (2-3줄)
```

```
    - 상세 설명
    - 다음에 물어볼 만한 질문 3개
    """

    # 응답 생성 및 후처리
    state['final_response'] = response
    return state
```

2.3 LangGraph 워크플로우 구성

python

```python
class CitizenServiceWorkflow:
    """시민 서비스 Multi-Agent 워크플로우"""

    def __init__(self):
        self.graph = StateGraph(AgentState)
        self.setup_workflow()

    def setup_workflow(self):
        """워크플로우 그래프 구성"""

        # Agent 노드 추가
        self.graph.add_node("analyze_query", self.analyze_query_node)
        self.graph.add_node("retrieve_docs", self.retrieve_docs_node)
        self.graph.add_node("process_content", self.process_content_node)
        self.graph.add_node("generate_response", self.generate_response_node)

        # 엣지 연결
        self.graph.add_edge("analyze_query", "retrieve_docs")
        self.graph.add_edge("retrieve_docs", "process_content")
        self.graph.add_edge("process_content", "generate_response")
        self.graph.add_edge("generate_response", END)

        # 시작점 설정
        self.graph.set_entry_point("analyze_query")

        # 조건부 라우팅 (필요시)
        self.graph.add_conditional_edges(
            "retrieve_docs",
            self.should_retry_search,
            {
                "retry": "analyze_query",
                "continue": "process_content",
                "insufficient": "generate_response"
            }
        )

    def should_retry_search(self, state: AgentState) -> str:
        """검색 결과에 따른 라우팅 결정"""
        if not state['retrieved_documents']:
            return "insufficient"
        elif len(state['retrieved_documents']) < 2:
            return "retry"
        else:
            return "continue"
```

3. API 설계

3.1 RESTful API 명세

3.1.1 검색 API

```python
```

```python
# 검색 관련 API
from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel
from typing import List, Optional

router = APIRouter(prefix="/api/v1/search", tags=["search"])

class SearchRequest(BaseModel):
    query: str
    category: Optional[str] = None
    max_results: int = 5
    include_similar: bool = True
    user_context: Optional[dict] = None

class DocumentResult(BaseModel):
    id: str
    title: str
    summary: str
    content_preview: str
    category: str
    publish_date: str
    relevance_score: float
    source_url: str
    metadata: dict

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

@router.post("/query", response_model=SearchResponse)
async def search_documents(
    request: SearchRequest,
    workflow_service: WorkflowService = Depends()
):
    """
    자연어 질의를 통한 문서 검색

    - **query**: 사용자의 자연어 질문
    - **category**: 검색 범위를 제한할 카테고리 (선택)
    - **max_results**: 반환할 최대 결과 수
    - **include_similar**: 유사 질문 포함 여부
```

```
    """
    try:
        result = await workflow_service.execute_search_workflow(
            query=request.query,
            category=request.category,
            max_results=request.max_results,
            user_context=request.user_context
        )
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/categories")
async def get_categories():
    """사용 가능한 카테고리 목록 조회"""
    return {
        "categories": [
            {"id": "welfare", "name": "복지", "icon": "🎁"},
            {"id": "tax", "name": "세금", "icon": "💰"},
            {"id": "education", "name": "교육", "icon": "📚"},
            {"id": "housing", "name": "주택", "icon": "🏠"},
            {"id": "employment", "name": "취업", "icon": "💼"},
            {"id": "business", "name": "사업", "icon": "🏢"},
            {"id": "health", "name": "보건", "icon": "⚕️"},
            {"id": "environment", "name": "환경", "icon": "🌱"}
        ]
    }

@router.get("/popular")
async def get_popular_queries():
    """인기 검색어 조회"""
    return {
        "popular_queries": [
            "아이 양육비 지원 신청",
            "실업급여 받는 방법",
            "주택청약 신청 조건",
            "소상공인 대출 지원",
            "국민연금 가입 방법"
        ]
    }
```

3.1.2 대화형 상담 API

python

```python
# 채팅 관련 API
class ChatMessage(BaseModel):
    content: str
    timestamp: datetime
    sender: str  # "user" or "assistant"
    metadata: Optional[dict] = None

class ChatRequest(BaseModel):
    message: str
    session_id: str
    context: Optional[dict] = None

class ChatResponse(BaseModel):
    response: str
    session_id: str
    suggested_questions: List[str]
    related_documents: List[DocumentResult]
    conversation_summary: Optional[str] = None
    next_steps: List[str]

@router.post("/chat/message", response_model=ChatResponse)
async def send_chat_message(
    request: ChatRequest,
    workflow_service: WorkflowService = Depends()
):
    """
    대화형 메시지 처리

    - **message**: 사용자 메시지
    - **session_id**: 대화 세션 ID
    - **context**: 추가 컨텍스트 정보
    """
    try:
        result = await workflow_service.execute_chat_workflow(
            message=request.message,
            session_id=request.session_id,
            context=request.context
        )
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/chat/history/{session_id}")
async def get_chat_history(
    session_id: str,
    limit: int = 50,
```

```
    session_service: SessionService = Depends()
):
    """대화 기록 조회"""
    try:
        history = await session_service.get_chat_history(session_id, limit)
        return {"history": history}
    except Exception as e:
        raise HTTPException(status_code=404, detail="Session not found")

@router.post("/chat/session")
async def create_chat_session(
    session_service: SessionService = Depends()
):
    """새 채팅 세션 생성"""
    session_id = await session_service.create_session()
    return {"session_id": session_id}
```

3.2 API 미들웨어 및 보안

python

```python
# 미들웨어 설정
from fastapi.middleware.cors import CORSMiddleware
from fastapi.middleware.trustedhost import TrustedHostMiddleware
import time

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:8501"],  # Streamlit 클라이언트
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    """응답 시간 측정 미들웨어"""
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response

# 요청 제한 및 검증
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

@router.post("/search/query")
@limiter.limit("30/minute")  # 분당 30회 제한
async def search_documents_rate_limited(request: Request, ...):
    pass
```

4. 데이터베이스 설계

4.1 Vector Database 스키마

python

```python
# FAISS Vector DB 구조
class DocumentVector:
    """문서 벡터 표현"""
    def __init__(self):
        self.vector_id: str          # 고유 벡터 ID
        self.document_id: str        # 원본 문서 ID
        self.chunk_id: str           # 청크 ID (문서 분할 단위)
        self.embedding: np.ndarray   # 768차원 임베딩 벡터
        self.metadata: dict          # 메타데이터

class DocumentMetadata:
    """문서 메타데이터"""
    title: str                  # 문서 제목
    category: str               # 카테고리 (welfare, tax, education 등)
    subcategory: str            # 하위 카테고리
    publish_date: datetime      # 발행일
    update_date: datetime       # 수정일
    source_url: str             # 원본 URL
    department: str             # 발행 부처
    document_type: str          # 문서 유형 (법령, 고시, 공고 등)
    keywords: List[str]         # 키워드 태그
    citizen_relevance: float    # 시민 연관성 점수 (0-1)
    complexity_level: int       # 복잡도 (1-5)
    life_stage: List[str]       # 생애주기 (출생, 교육, 취업, 결혼, 육아, 은퇴 등)
    target_audience: List[str] # 대상 (개인, 사업자, 특정계층 등)
```

4.2 세션 및 캐시 데이터베이스

python

```python
# SQLite 기반 세션 관리
class ChatSession(BaseModel):
    session_id: str = Field(primary_key=True)
    created_at: datetime
    last_activity: datetime
    user_context: Optional[dict] = None
    conversation_summary: Optional[str] = None
    message_count: int = 0

class ChatMessage(BaseModel):
    message_id: str = Field(primary_key=True)
    session_id: str = Field(foreign_key="chat_session.session_id")
    content: str
    sender: str  # "user" or "assistant"
    timestamp: datetime
    metadata: Optional[dict] = None

class SearchHistory(BaseModel):
    search_id: str = Field(primary_key=True)
    session_id: str = Field(foreign_key="chat_session.session_id")
    query: str
    category: Optional[str] = None
    results_count: int
    timestamp: datetime
    user_satisfaction: Optional[int] = None  # 1-5 점수

# Redis 캐시 구조
class CacheKey:
    SEARCH_RESULTS = "search:{query_hash}"
    DOCUMENT_CONTENT = "doc:{doc_id}"
    USER_SESSION = "session:{session_id}"
    POPULAR_QUERIES = "popular:queries"
    CATEGORY_STATS = "stats:category"
```

5. 프론트엔드 상세 설계

5.1 Streamlit 앱 구조

python

```python
# main.py - 메인 애플리케이션
import streamlit as st
from config.settings import Settings
from services.api_client import ApiClient
from utils.session_manager import SessionManager

def main():
    st.set_page_config(
        page_title="정부 정책 도우미",
        page_icon="🏛️",
        layout="wide",
        initial_sidebar_state="collapsed"
    )

    # 커스텀 CSS 로드
    load_custom_css()

    # 세션 초기화
    session_manager = SessionManager()
    session_manager.initialize_session()

    # 페이지 라우팅
    page = st.sidebar.selectbox(
        "페이지 선택",
        ["🏠 홈", "🔍 검색", "💬 상담", "📁 카테고리", "📋 기록"]
    )

    if page == "🏠 홈":
        render_home_page()
    elif page == "🔍 검색":
        render_search_page()
    elif page == "💬 상담":
        render_chat_page()
    elif page == "📁 카테고리":
        render_category_page()
    elif page == "📋 기록":
        render_history_page()

def load_custom_css():
    """커스텀 CSS 스타일 로드"""
    st.markdown("""
    <style>
    .main-header {
        font-size: 2.5rem;
        font-weight: bold;
        color: #1f4e79;
```

```
    text-align: center;
    margin-bottom: 2rem;
}

.search-box {
    border-radius: 25px;
    border: 2px solid #e1e5e9;
    padding: 15px 20px;
    font-size: 1.1rem;
    width: 100%;
}

.result-card {
    background: white;
    border-radius: 10px;
    padding: 20px;
    margin: 10px 0;
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    border-left: 4px solid #1f4e79;
}

.category-button {
    background: linear-gradient(45deg, #667eea 0%, #764ba2 100%);
    color: white;
    border: none;
    border-radius: 15px;
    padding: 20px;
    margin: 10px;
    font-size: 1.1rem;
    cursor: pointer;
    transition: transform 0.2s;
}

.chat-bubble-user {
    background: #dcf8c6;
    border-radius: 18px 18px 4px 18px;
    padding: 12px 16px;
    margin: 8px 0;
    max-width: 80%;
    margin-left: auto;
}

.chat-bubble-assistant {
    background: #f1f1f2;
    border-radius: 18px 18px 18px 4px;
    padding: 12px 16px;
    margin: 8px 0;
```

```
    max-width: 80%;
}
</style>
""", unsafe_allow_html=True)
```

5.2 핵심 컴포넌트 설계

5.2.1 검색 인터페이스

python

```python
# components/search_interface.py
class SearchInterface:
    def __init__(self, api_client: ApiClient):
        self.api_client = api_client

    def render_search_box(self):
        """메인 검색 입력 박스"""
        col1, col2, col3 = st.columns([1, 6, 1])

        with col2:
            st.markdown('<div class="main-header">무엇을 도와드릴까요?</div>',
                        unsafe_allow_html=True)

            # 검색 입력
            search_query = st.text_input(
                "",
                placeholder="예: 아이 양육비 지원은 어떻게 받나요?",
                key="main_search",
                label_visibility="collapsed"
            )

            # 카테고리 필터
            categories = ["전체", "복지", "세금", "교육", "주택", "취업", "사업"]
            selected_category = st.selectbox(
                "분야 선택",
                categories,
                key="category_filter"
            )

            # 검색 버튼
            if st.button("🔍 검색하기", key="search_btn", type="primary"):
                if search_query.strip():
                    with st.spinner("검색 중..."):
                        results = self.search_documents(search_query, selected_category)
                        self.display_results(results)
                else:
                    st.warning("검색어를 입력해주세요.")

    def render_popular_queries(self):
        """인기 검색어 표시"""
        st.subheader("🔥 인기 검색어")
        popular_queries = self.api_client.get_popular_queries()

        cols = st.columns(3)
        for i, query in enumerate(popular_queries[:6]):
            with cols[i % 3]:
```

```
            if st.button(f"#{query}", key=f"popular_{i}"):
                st.session_state.main_search = query
                st.experimental_rerun()

    async def search_documents(self, query: str, category: str = None):
        """문서 검색 실행"""
        try:
            response = await self.api_client.search_documents(
                query=query,
                category=category if category != "전체" else None,
                max_results=5
            )
            return response
        except Exception as e:
            st.error(f"검색 중 오류가 발생했습니다: {str(e)}")
            return None

    def display_results(self, search_response):
        """검색 결과 표시"""
        if not search_response or not search_response.results:
            st.info("검색 결과가 없습니다. 다른 키워드로 시도해보세요.")
            return

        st.success(f"총 {search_response.total_count}개의 관련 문서를 찾았습니다.")

        for i, result in enumerate(search_response.results):
            with st.container():
                st.markdown(f"""
                <div class="result-card">
                    <h3>📄 {result.title}</h3>
                    <p><strong>📝 요약:</strong> {result.summary}</p>
                    <p><strong>🏷️ 분야:</strong> {result.category}</p>
                    <p><strong>📅 발행일:</strong> {result.publish_date}</p>
                    <p><strong>🎯 관련성:</strong> {result.relevance_score:.1%}</p>
                </div>
                """, unsafe_allow_html=True)

                # 상세 보기 버튼
                if st.button(f"자세히 보기", key=f"detail_{i}"):
                    self.show_document_detail(result)

                st.markdown("---")
```

5.2.2 대화형 인터페이스

python

```python
# components/chat_interface.py
class ChatInterface:
    def __init__(self, api_client: ApiClient):
        self.api_client = api_client

    def render_chat_page(self):
        """대화형 상담 페이지"""
        st.title("💬 AI 정책 상담사")
        st.markdown("궁금한 정책에 대해 자유롭게 질문하세요!")

        # 세션 초기화
        if "chat_session_id" not in st.session_state:
            st.session_state.chat_session_id = self.create_new_session()

        if "chat_messages" not in st.session_state:
            st.session_state.chat_messages = []
            # 환영 메시지
            welcome_msg = {
                "sender": "assistant",
                "content": "안녕하세요! 정부 정책에 대해 궁금한 것이 있으시면 언제든 물어보세요. 😊",
                "timestamp": datetime.now()
            }
            st.session_state.chat_messages.append(welcome_msg)

        # 채팅 기록 표시
        self.display_chat_history()

        # 메시지 입력
        self.render_message_input()

        # 추천 질문 버튼
        self.render_suggested_questions()

    def display_chat_history(self):
        """채팅 기록 표시"""
        chat_container = st.container()

        with chat_container:
            for message in st.session_state.chat_messages:
                if message["sender"] == "user":
                    st.markdown(f"""
                    <div class="chat-bubble-user">
                        {message["content"]}
                        <small style="color: #666; font-size: 0.8em;">
                            {message["timestamp"].strftime("%H:%M")}
                        </small>
```

```
                </div>
                """, unsafe_allow_html=True)
            else:
                st.markdown(f"""
                <div class="chat-bubble-assistant">
                    🤖 {message["content"]}
                    <small style="color: #666; font-size: 0.8em;">
                        {message["timestamp"].strftime("%H:%M")}
                    </small>
                </div>
                """, unsafe_allow_html=True)

    def render_message_input(self):
        """메시지 입력 인터페이스"""
        with st.form(key="chat_form", clear_on_submit=True):
            col1, col2 = st.columns([5, 1])

            with col1:
                user_input = st.text_input(
                    "",
                    placeholder="정책에 대해 궁금한 점을 입력하세요...",
                    label_visibility="collapsed"
                )

            with col2:
                send_button = st.form_submit_button("전송", type="primary")

            if send_button and user_input.strip():
                self.handle_user_message(user_input.strip())

    async def handle_user_message(self, message: str):
        """사용자 메시지 처리"""
        # 사용자 메시지 추가
        user_message = {
            "sender": "user",
            "content": message,
            "timestamp": datetime.now()
        }
        st.session_state.chat_messages.append(user_message)

        # AI 응답 생성
        with st.spinner("답변을 생성하고 있습니다..."):
            try:
                response = await self.api_client.send_chat_message(
                    message=message,
                    session_id=st.session_state.chat_session_id
                )
```

```
            # AI 응답 추가
            ai_message = {
                "sender": "assistant",
                "content": response.response,
                "timestamp": datetime.now(),
                "suggestions": response.suggested_questions,
                "related_docs": response.related_documents
            }
            st.session_state.chat_messages.append(ai_message)

            # 페이지 새로고침
            st.experimental_rerun()

        except Exception as e:
            st.error(f"답변 생성 중 오류가 발생했습니다: {str(e)}")

    def render_suggested_questions(self):
        """추천 질문 표시"""
        if st.session_state.chat_messages:
            last_message = st.session_state.chat_messages[-1]
            if (last_message["sender"] == "assistant" and
                "suggestions" in last_message):

                st.subheader("💡 이런 질문은 어떠세요?")
                for i, suggestion in enumerate(last_message["suggestions"]):
                    if st.button(suggestion, key=f"suggestion_{i}"):
                        self.handle_user_message(suggestion)
```

6. 배포 및 운영 설계

6.1 Docker 컨테이너 설정

dockerfile

```
# Dockerfile.server (FastAPI 서버)
FROM python:3.11-slim

WORKDIR /app

# 시스템 의존성 설치
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Python 의존성 설치
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 소스 코드 복사
COPY fastapi_server/ ./fastapi_server/
COPY .env .

# 환경변수 설정
ENV PYTHONPATH=/app
ENV UVICORN_HOST=0.0.0.0
ENV UVICORN_PORT=8000

# 포트 노출
EXPOSE 8000

# 서버 실행
CMD ["uvicorn", "fastapi_server.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

dockerfile

```
# Dockerfile.client (Streamlit 클라이언트)
FROM python:3.11-slim

WORKDIR /app

# 시스템 의존성 설치
RUN apt-get update && apt-get install -y \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Python 의존성 설치
COPY requirements-client.txt .
RUN pip install --no-cache-dir -r requirements-client.txt

# 소스 코드 복사
COPY streamlit_app/ ./streamlit_app/
COPY .env .

# 환경변수 설정
ENV PYTHONPATH=/app

# 포트 노출
EXPOSE 8501

# Streamlit 실행
CMD ["streamlit", "run", "streamlit_app/main.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

6.2 Docker Compose 설정

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  # FastAPI 서버
  api-server:
    build:
      context: .
      dockerfile: Dockerfile.server
    ports:
      - "8000:8000"
    environment:
      - AOAI_ENDPOINT=${AOAI_ENDPOINT}
      - AOAI_API_KEY=${AOAI_API_KEY}
      - AOAI_DEPLOY_GPT4O_MINI=${AOAI_DEPLOY_GPT4O_MINI}
      - AOAI_DEPLOY_GPT4O=${AOAI_DEPLOY_GPT4O}
      - AOAI_DEPLOY_EMBED_3_LARGE=${AOAI_DEPLOY_EMBED_3_LARGE}
      - VECTOR_DB_PATH=/app/data/vector_db
      - LOG_LEVEL=INFO
    volumes:
      - ./data:/app/data
      - ./logs:/app/logs
    depends_on:
      - vector-db
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  # Streamlit 클라이언트
  web-client:
    build:
      context: .
      dockerfile: Dockerfile.client
    ports:
      - "8501:8501"
    environment:
      - SERVER_URL=http://api-server:8000
    depends_on:
      - api-server
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8501/_stcore/health"]
      interval: 30s
```

```
    timeout: 10s
    retries: 3

  # Vector Database (ChromaDB)
  vector-db:
    image: chromadb/chroma:latest
    ports:
      - "8002:8000"
    volumes:
      - ./data/chromadb:/chroma/chroma
    environment:
      - CHROMA_SERVER_HOST=0.0.0.0
      - CHROMA_SERVER_PORT=8000
    restart: unless-stopped

  # Redis 캐시 (선택사항)
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    restart: unless-stopped
    command: redis-server --appendonly yes

volumes:
  redis_data:
```

6.3 환경 설정 관리

python

```python
# config/settings.py
from pydantic import BaseSettings
from typing import Optional
import os

class Settings(BaseSettings):
    """애플리케이션 설정"""

    # Azure OpenAI 설정
    aoai_endpoint: str
    aoai_api_key: str
    aoai_deploy_gpt4o_mini: str
    aoai_deploy_gpt4o: str
    aoai_deploy_embed_3_large: str

    # 서버 설정
    server_host: str = "0.0.0.0"
    server_port: int = 8000
    client_url: str = "http://localhost:8501"

    # 데이터베이스 설정
    vector_db_path: str = "./data/vector_db"
    session_db_path: str = "./data/sessions.db"

    # 캐시 설정
    redis_url: Optional[str] = None
    cache_ttl: int = 3600  # 1시간

    # 로깅 설정
    log_level: str = "INFO"
    log_file: str = "./logs/app.log"

    # API 설정
    api_rate_limit: str = "30/minute"
    max_search_results: int = 10
    search_timeout: int = 30

    # Vector DB 설정
    embedding_dimension: int = 768
    max_chunk_size: int = 500
    chunk_overlap: int = 50

    class Config:
        env_file = ".env"
        case_sensitive = False
```

```python
# 싱글톤 설정 인스턴스
settings = Settings()
```

7. 모니터링 및 로깅 설계

7.1 로깅 시스템

python

```python
# utils/logging.py
import logging
import sys
from datetime import datetime
from pathlib import Path
import json

class StructuredLogger:
    """구조화된 로깅 시스템"""

    def __init__(self, name: str, log_file: str = None):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.INFO)

        # 콘솔 핸들러
        console_handler = logging.StreamHandler(sys.stdout)
        console_handler.setLevel(logging.INFO)

        # 파일 핸들러
        if log_file:
            Path(log_file).parent.mkdir(parents=True, exist_ok=True)
            file_handler = logging.FileHandler(log_file)
            file_handler.setLevel(logging.DEBUG)

            # JSON 포맷터
            formatter = JsonFormatter()
            file_handler.setFormatter(formatter)
            console_handler.setFormatter(formatter)

            self.logger.addHandler(file_handler)

        self.logger.addHandler(console_handler)

    def log_search_request(self, query: str, category: str, user_id: str = None):
        """검색 요청 로깅"""
        self.logger.info("search_request", extra={
            "event_type": "search_request",
            "query": query,
            "category": category,
            "user_id": user_id,
            "timestamp": datetime.now().isoformat()
        })

    def log_search_results(self, query: str, results_count: int,
                           processing_time: float, confidence: float):
        """검색 결과 로깅"""
```

```
        self.logger.info("search_results", extra={
            "event_type": "search_results",
            "query": query,
            "results_count": results_count,
            "processing_time": processing_time,
            "confidence_score": confidence,
            "timestamp": datetime.now().isoformat()
        })

    def log_error(self, error: Exception, context: dict = None):
        """에러 로깅"""
        self.logger.error("application_error", extra={
            "event_type": "error",
            "error_type": type(error).__name__,
            "error_message": str(error),
            "context": context or {},
            "timestamp": datetime.now().isoformat()
        })

class JsonFormatter(logging.Formatter):
    """JSON 로그 포맷터"""

    def format(self, record):
        log_entry = {
            "timestamp": datetime.fromtimestamp(record.created).isoformat(),
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
        }

        # 추가 필드
        if hasattr(record, 'event_type'):
            log_entry.update(record.__dict__)
            log_entry.pop('name', None)
            log_entry.pop('msg', None)
            log_entry.pop('args', None)
            log_entry.pop('levelname', None)
            log_entry.pop('levelno', None)
            log_entry.pop('pathname', None)
            log_entry.pop('filename', None)
            log_entry.pop('module', None)
            log_entry.pop('lineno', None)
            log_entry.pop('funcName', None)
            log_entry.pop('created', None)
            log_entry.pop('msecs', None)
            log_entry.pop('relativeCreated', None)
            log_entry.pop('thread', None)
```

```
        log_entry.pop('threadName', None)
        log_entry.pop('processName', None)
        log_entry.pop('process', None)

        return json.dumps(log_entry, ensure_ascii=False)
```

7.2 성능 모니터링

```python
```

```python
# utils/monitoring.py
import time
import psutil
import asyncio
from functools import wraps
from typing import Callable
import prometheus_client

class PerformanceMonitor:
    """성능 모니터링 시스템"""

    def __init__(self):
        # Prometheus 메트릭 설정
        self.request_duration = prometheus_client.Histogram(
            'request_duration_seconds',
            'Request duration',
            ['method', 'endpoint']
        )

        self.request_count = prometheus_client.Counter(
            'requests_total',
            'Total requests',
            ['method', 'endpoint', 'status']
        )

        self.search_accuracy = prometheus_client.Histogram(
            'search_accuracy_score',
            'Search result accuracy',
            buckets=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
        )

    def measure_performance(self, operation: str):
        """성능 측정 데코레이터"""
        def decorator(func: Callable):
            @wraps(func)
            async def async_wrapper(*args, **kwargs):
                start_time = time.time()
                try:
                    result = await func(*args, **kwargs)
                    duration = time.time() - start_time

                    self.request_duration.labels(
                        method="async",
                        endpoint=operation
                    ).observe(duration)
```

```
                return result
            except Exception as e:
                duration = time.time() - start_time
                self.request_count.labels(
                    method="async",
                    endpoint=operation,
                    status="error"
                ).inc()
                raise e

        @wraps(func)
        def sync_wrapper(*args, **kwargs):
            start_time = time.time()
            try:
                result = func(*args, **kwargs)
                duration = time.time() - start_time

                self.request_duration.labels(
                    method="sync",
                    endpoint=operation
                ).observe(duration)

                return result
            except Exception as e:
                self.request_count.labels(
                    method="sync",
                    endpoint=operation,
                    status="error"
                ).inc()
                raise e

        if asyncio.iscoroutinefunction(func):
            return async_wrapper
        else:
            return sync_wrapper

    return decorator

def get_system_metrics(self) -> dict:
    """시스템 메트릭 수집"""
    return {
        "cpu_percent": psutil.cpu_percent(),
        "memory_percent": psutil.virtual_memory().percent,
        "disk_usage": psutil.disk_usage('/').percent,
        "active_connections": len(psutil.net_connections()),
        "timestamp": time.time()
    }
```

```python
# 사용 예시
monitor = PerformanceMonitor()

@monitor.measure_performance("document_search")
async def search_documents(query: str) -> dict:
    # 검색 로직
    pass
```

이 설계문서는 요구사항 정의서를 바탕으로 구체적인 시스템 구현 방안을 제시합니다. 특히 시민 친화적 서비스 구현에 중점을 두어 실제 사용 가능한 AI Agent 시스템을 구축할 수 있도록 상세한 기술적 가이드라인을 제공했습니다.