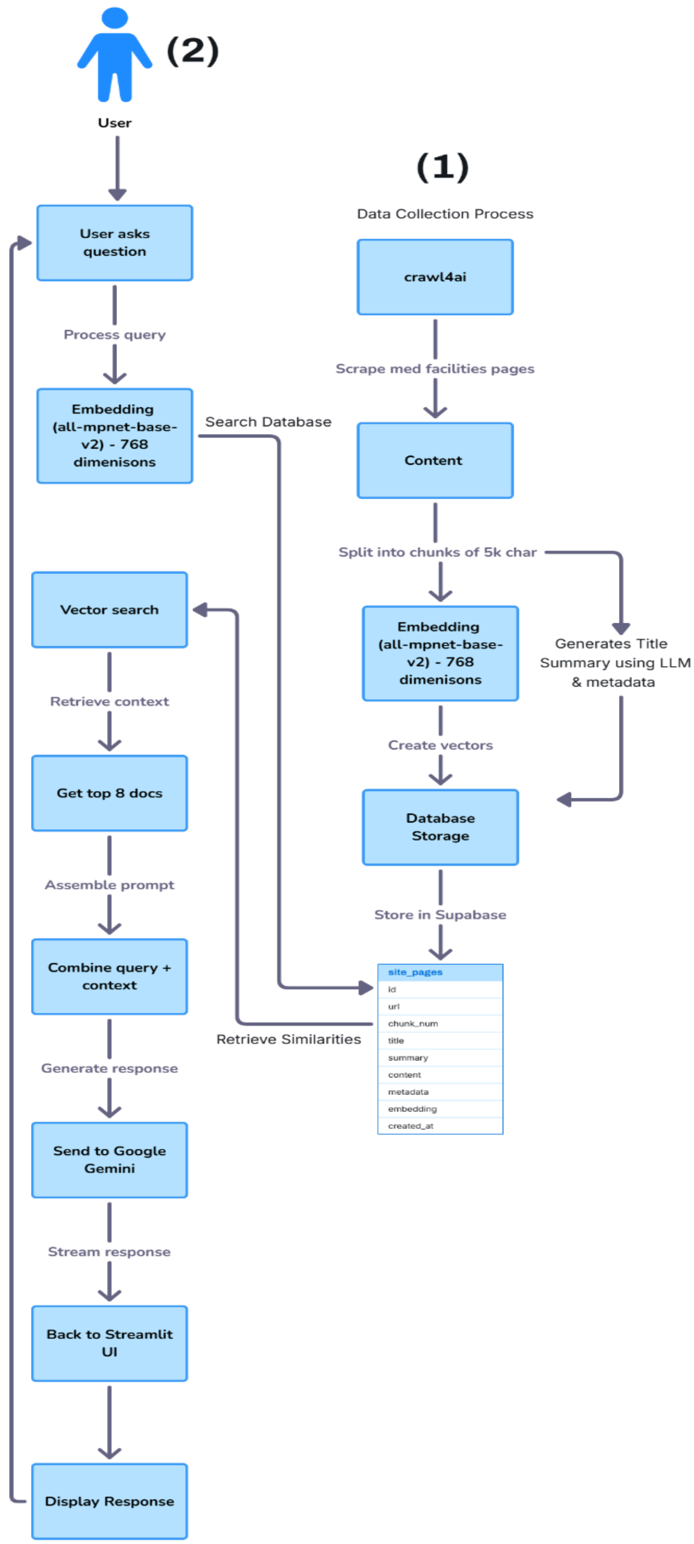
Chatbot - System Design



1. Data Collection Phase (Web Scraping)

File: crawl_stanford_medical_facilities.py

The workflow starts with web scraping using Crawl4AI:

```
from crawl4ai import AsyncWebCrawler, BrowserConfig, CrawlerRunConfig, CacheMode
```

Target URLs: The system crawls 6 specific Stanford Medical Facilities pages:
Emergency Management
Safety Guidelines & Resources
About Us
Projects
Space Planning & Assets
Locations

Crawling Process:

1. Async Web Crawler: Uses headless browser with rate limiting (for api bottlenecks)
2. Content Extraction: Converts HTML to markdown using markdown_v2.raw_markdown
3. Chunking: Splits content into 5000-character chunks while preserving:
Code blocks (```)
Paragraph boundaries (\n\n)
Sentence boundaries (.)

2. Content Processing & Embedding Generation

Chunk Processing Pipeline:

```
def chunk_text(text: str, chunk_size: int = 5000) -> List[str]:
    """Split text into chunks, respecting code blocks and paragraphs."""
```

For each chunk, the system:

1. Generates Title & Summary: Uses Gemini 2.0 to extract descriptive titles and summaries
2. Creates Embeddings: Uses all-mpnet-base-v2 model (768 dimensions)
3. Builds Metadata: Includes source, chunk size, timestamp, URL path

3. Vector Database Storage

Database Schema (site_pages.sql):

```
create table site_pages (
    id bigserial primary key,
```

```
    url varchar not null,
    chunk_number integer not null,
    title varchar not null,
    summary varchar not null,
    content text not null,
    metadata jsonb not null default '{}'::jsonb,
    embedding vector(768),
    created_at timestamp with time zone default timezone('utc'::text, now()) not null,
```

Key Features:

- pgvector Extension: Enables vector similarity search
- Cosine Similarity: Uses vector_cosine_ops for similarity matching
- RLS Policies: Public read access for security

4. RAG Query Processing

File: stanford_medical_facilities_expert.py
When a user asks a question, the RAG process works as follows:

Step 1: Query Embedding

```
def get_embedding(text: str) -> List[float]:
    """Get embedding vector using all-mpnet-base-v2."""
    try:
        embedding = embedding_model.encode(text)
        return embedding.tolist()
```

Step 2: Vector Similarity Search

```
async def retrieve_relevant_documentation(user_query: str) -> tuple[str, List[Dict]]:
    """Retrieve relevant documentation chunks based on the query with RAG."""
    try:
        # Get the embedding for the query
        query_embedding = get_embedding(user_query)

        # Query Supabase for relevant documents
        result = supabase.rpc(
            'match_site_pages',
            {
                'query_embedding': query_embedding,
                'match_count': 8,  # Increased to get more context
                'filter': {'source': 'stanford_medical_facilities'}
            }
```

```
    ).execute()
```

Step 3: Context Assembly

- Retrieves top 8 most similar chunks
- Formats chunks with titles and content
- Collects source URLs for attribution

5. Response Generation

LLM Integration:

```
async def generate_response(user_query: str) -> str:
  """Generate a response using Gemini with RAG."""
  try:
    # First, retrieve relevant documentation
    relevant_docs, source_urls = await retrieve_relevant_documentation(user_query)

    # Get list of available pages
    available_pages = await list_documentation_pages()

    # Create the prompt for Gemini
    system_prompt = """You are a helpful and knowledgeable assistant for Stanford
Medical Facilities..."""
```

Prompt Engineering:

- System prompt defines the assistant's role
- Includes retrieved documentation chunks
- Lists available pages for context
- Requests source URL attribution

6. User Interface

File: streamlit_ui.py
Streamlit Web Interface:

- Chat Interface: User-friendly chat UI
- Streaming Responses: Real-time text streaming
- Message History: Persistent conversation memory
- Medical Branding: Stanford-themed styling

Key Features:

```
async def run_agent_with_streaming(user_input: str):
    """
    Run the agent with streaming text for the user_input prompt.
    """
    # Run the agent and get the result
    result = await stanford_medical_facilities_expert.run_stream(user_input)

    # We'll gather partial text to show incrementally
    partial_text = ""
    message_placeholder = st.empty()

    # Render partial text as it arrives
    async for chunk in result.stream_text(delta=True):
        partial_text += chunk
        message_placeholder.markdown(partial_text)
```

Complete Workflow Summary

1. Data Ingestion: Crawl4AI scrapes Stanford Medical Facilities websites
2. Content Processing: Chunk text, generate embeddings, extract metadata
3. Vector Storage: Store in Supabase with pgvector for similarity search
4. Query Processing: User question → embedding → similarity search → context retrieval
5. Response Generation: Gemini 2.0 generates response using retrieved context
6. User Interface: Streamlit provides interactive chat interface

Key Technologies:

Crawl4AI: Web scraping
Sentence Transformers: Embedding generation
Supabase + pgvector: Vector database
Google Gemini 2.0: LLM for response generation
Streamlit: Web interface