AI Chat UI: Beyond ChatGPT

Building Advanced Chat Interfaces with FastAPI

Streaming, Context Management, and Business Integration

Module Overview

What You'll Learn

- **Streaming responses** for real-time interactivity
- **Context management** with conversation history
- **Internet search integration** with smart triggering
- **Database connectivity** for business context
- **Advanced features** for production applications

The Problem with Basic Chat

Current Limitations

- **No real-time feedback** - Users wait for complete responses
- **Poor context management** - Limited conversation memory
- **No external data** - Can't access current information
- **Isolated conversations** - No business context
- **Basic functionality** - Missing advanced features

Our Solution: Advanced Chat UI

Key Features

- **Streaming responses** - Real-time feedback
- **Smart context management** - Last 20 messages
- **Internet search integration** - Current information access
- **Business context** - Client and project integration
- **Extensible architecture** - Easy to enhance

1. Streaming Responses

Why Streaming?

- **Real-time feedback** - Users see progress
- **Reduced perceived latency** - Feels faster
- **Better user experience** - More interactive
- **Professional feel** - Like ChatGPT/Claude

Streaming Implementation

Technical Approach

```python
# FastAPI EventSourceResponse
@router.post("/chat/stream")
async def chat_with_llama_stream(request: Request):
    async def generate():
        async for chunk in openai_stream:
            yield f"data: {chunk}\n\n"

    return EventSourceResponse(generate())
```

Frontend processes `data:` events for real-time updates

2. Context Management

Smart Context Strategy

- **Last 20 messages** - Balances continuity with performance
- **Chronological order** - Maintains conversation flow
- **Token management** - Stays within LLM limits
- **Database storage** - Persistent conversation history

Context Implementation

Database Design

```
class Message(SQLModel, table=True):
    id: UUID
    conversation_id: UUID
    role: str  # user, assistant, system
    content: str
    created_at: datetime
```

Smart retrieval with `get_conversation_context()`

3. Internet Search Integration

Current Implementation

Keyword-Based Triggering:

- Detects current events keywords
- Keywords: "latest", "current", "2024", "news", "price"
- Uses Tavily API for structured results
- LLM-optimized search responses

Search Triggering Strategies

5 Approaches

1. **Keyword matching** (current) - Simple and reliable
2. **Intent classification** - ML-based detection
3. **LLM-based decisions** - AI determines need
4. **Hybrid approach** - Multiple signals
5. **User-controlled** - Manual search button

Strategy 1: Keyword Matching

Current Implementation

```
SEARCH_KEYWORDS = ["latest", "current", "2024", "news", "price", "weather"]

def should_search_web(user_message: str) -> bool:
    return any(keyword in user_message.lower()
               for keyword in SEARCH_KEYWORDS)
```

Simple, fast, and reliable

Strategy 2: Intent Classification

ML-Based Approach

```
def classify_search_intent(user_message: str) -> bool:
    # Train a model to detect when users need current information
    # More sophisticated than keyword matching
    pass
```

More accurate but requires training data

Strategy 3: LLM-Based Decisions

AI-Powered Detection

```
def should_search_web(user_message: str, llm_response: str) -> bool:
    # LLM analyzes if it needs current data to answer properly
    # More context-aware than keyword detection
    pass
```

Most intelligent but slower

Strategy 4: Hybrid Approach

Multiple Signals

```
def should_search(user_message: str, conversation_history: list) -> bool:
    keyword_match = any(keyword in user_message.lower()
                        for keyword in SEARCH_KEYWORDS)
    intent_score = classify_intent(user_message)
    llm_confidence = get_llm_confidence(user_message)

    return keyword_match or (intent_score > 0.7) or (llm_confidence < 0.5)
```

Best of all worlds

Strategy 5: User-Controlled

Manual Search Button

- **"Search Web" button** in UI
- **User explicitly requests** web search
- **Most reliable** but requires interaction
- **Good for power users**

4. Database Integration

Hybrid Search Integration

Current Capabilities:

- Vector database (FAISS) + Full-text search (FTS5)
- BM25 ranking algorithm
- Available in search page
- Can be integrated into chat

Database Integration Strategies

3 Approaches

1. **Automatic context retrieval** - AI finds relevant data
2. **User-triggered search** - Manual knowledge base search
3. **Smart context injection** - Domain-specific context

Strategy A: Automatic Context Retrieval

AI-Powered Data Access

```python
def get_relevant_context(user_message: str, conversation_id: str) -> str:
    # Search for relevant documents/conversations
    search_results = hybrid_search_service.search(user_message, limit=5)

    # Format results for LLM context
    context = format_search_results_for_llm(search_results)
    return context
```

Seamless but can be unpredictable

Strategy B: User-Triggered Search

Manual Knowledge Base Access

- **"Search Knowledge Base" button**
- **User explicitly searches** their data
- **Most reliable** but requires interaction
- **Good for specific queries**

Strategy C: Smart Context Injection

Domain-Specific Context

```
def smart_context_injection(user_message: str, conversation_history: list) -> str:
    if is_asking_about_customers(user_message):
        return get_customer_context(user_message)
    elif is_asking_about_projects(user_message):
        return get_project_context(user_message)
    # ... other domain-specific contexts
```

Intelligent and contextual

Customer Database Integration

Business Use Cases

- "Find customers similar to Acme Corp"
- "What products do our enterprise clients use?"
- "Show me customers who haven't been contacted in 6 months"

Customer Integration Implementation

Data Flow

1. **Data Preparation** - Convert customer data to embeddings
2. **Search Integration** - Use hybrid search to find relevant customers
3. **Context Formatting** - Format results for LLM consumption
4. **Response Enhancement** - LLM provides human-like responses with data

Customer Service Example

Implementation

```
class CustomerContextService:
    def get_customer_context(self, query: str) -> str:
        # Search customer database using hybrid search
        results = self.hybrid_search.search(query, content_type="customer")

        # Format for LLM
        context = "Relevant customers:\n"
        for customer in results:
            context += f"- {customer.name}: {customer.summary}\n"

        return context
```

Document Integration

Supported Document Types

- **PDFs** - Contracts, reports, manuals
- **Word documents** - Proposals, specifications
- **Text files** - Notes, procedures
- **Web pages** - Company knowledge base

Document Processing Pipeline

Step-by-Step

```
class DocumentProcessor:
    def process_document(self, file_path: str) -> List[Chunk]:
        # Extract text from document
        text = extract_text(file_path)

        # Split into chunks
        chunks = chunk_text(text)

        # Generate embeddings
        embeddings = generate_embeddings(chunks)

        # Store in database
        store_chunks(chunks, embeddings)
```

Learning Path

Phase 1: Core Functionality

1. **Study streaming implementation** (`services/chat_service.py`)
2. **Understand context management** (`services/chat_history_service.py`)
3. **Test different search triggering strategies**

Learning Path

Phase 2: Database Integration

1. **Integrate hybrid search** into chat responses
2. **Add customer database** connectivity
3. **Implement document processing** pipeline

Learning Path

Phase 3: Advanced Features

1. **Add intent classification** for smarter search triggering
2. **Implement multi-modal** document support
3. **Create domain-specific** context injection

Learning Path

Phase 4: Production Features

1. **Add conversation analytics** and user behavior tracking
2. **Implement conversation** summarization
3. **Add conversation** export and sharing

Architecture Overview

Current Architecture

```
User Input → Chat Service → OpenRouter API → Streaming Response
                ↓
        Chat History Service → Database Storage
                ↓
        Web Search Service → Tavily API (when triggered)
```

Enhanced Architecture

With Database Integration

```
User Input → Chat Service → OpenRouter API → Streaming Response
                ↓
        Chat History Service → Database Storage
                ↓
        Hybrid Search Service → Vector DB + FTS5
                ↓
        Document Service → Document Processing
                ↓
        Customer Service → Customer Database
```

Key Services

Service Architecture

- **ChatService** - Core chat functionality and LLM integration
- **WebSearchService** - Internet search with smart triggering
- **ChatHistoryService** - Conversation management and context
- **HybridSearchService** - Vector and text search capabilities
- **DocumentService** - Document processing and retrieval
- **CustomerService** - Customer data integration

Implementation Strategies

1. Gradual Enhancement

- Start with existing functionality
- Add one feature at a time
- Test and validate each addition
- Build on successful patterns

Implementation Strategies

2. User-Centric Design

- Focus on user experience
- Make features discoverable
- Provide clear feedback
- Allow user control

Implementation Strategies

3. Performance Considerations

- Cache frequently accessed data
- Optimize database queries
- Use async/await patterns
- Implement proper error handling

Advanced Concepts

Multi-Agent Systems

- **Different agents** for different tasks
- **Specialized search agents**
- **Document analysis agents**
- **Customer service agents**

Advanced Concepts

Context-Aware Responses

- Understand conversation history
- Maintain user preferences
- Adapt to user behavior
- Provide personalized responses

Advanced Concepts

Real-Time Collaboration

- **Multiple users** in same conversation
- **Real-time updates**
- **Conflict resolution**
- **Shared context**

Next Steps

Immediate Actions

1. **Experiment** with different search triggering strategies
2. **Integrate hybrid search** into chat responses
3. **Add document processing** capabilities
4. **Implement customer database** connectivity
5. **Test and optimize** performance
6. **Deploy** to production environment

Key Takeaways

What Makes This Superior

1. **Real-time streaming** - Better user experience
2. **Smart context management** - Maintains conversation flow
3. **External data integration** - Access to current information
4. **Business context** - Connects to your data
5. **Extensible architecture** - Easy to enhance

Ready to Build?

Start Learning

The best way to learn is by building!

Start with small enhancements and gradually add more sophisticated features.

Let's create something amazing! 🚀