**Streamlit UI Module: Detailed Technical Specifications**

**1. Functional Analysis**

**Purpose**

Interactive web interface for Project Citadel's RAG system, providing real-time chat-based interaction with the vectorized documentation knowledge base through a Streamlit-powered UI with streaming response capabilities.

**Core Capabilities**

**Real-Time Chat Interface:**

* **Conversational UI**: Chat-based interaction pattern similar to modern AI assistants
* **Message History**: Persistent conversation context across user sessions
* **Streaming Responses**: Real-time text generation with visual feedback
* **Multi-Message Support**: Handles complex conversation flows with tool calls

**RAG Integration:**

* **Seamless Agent Integration**: Direct connection to Pydantic AI RAG agent
* **Context Preservation**: Maintains conversation history for contextual responses
* **Tool Call Visualization**: Displays RAG retrieval operations and responses
* **Source Attribution**: Shows document sources for generated responses

**User Experience Features:**

* **Responsive Design**: Streamlit-native responsive layout
* **Visual Feedback**: Loading indicators and streaming cursors
* **Error Handling**: Graceful handling of agent failures
* **Session Management**: Stateful conversation persistence

**User Workflows**

**Primary Interaction Flow:**

1. User accesses Streamlit web interface
2. Interface initializes RAG agent dependencies
3. User enters question in chat input
4. System displays streaming response with visual feedback
5. Conversation history persists for context
6. User continues multi-turn conversation

**Advanced Features:**

* **Message Part Rendering**: Different display styles for user prompts, agent responses, tool calls
* **Async Streaming**: Non-blocking response generation
* **Session State Management**: Persistent conversation context

**2. Technical Specification**

**Architecture Components**

**Core Functions:**

1. **get\_agent\_deps() -> RAGDeps**

python

async def get\_agent\_deps():

return RAGDeps(

chroma\_client=get\_chroma\_client("./chroma\_db"),

collection\_name="docs",

embedding\_model="all-MiniLM-L6-v2"

)

* + **Purpose**: Initialize RAG agent dependencies
  + **Components**: ChromaDB client, collection configuration, embedding model
  + **Lifecycle**: Called once per session initialization

1. **display\_message\_part(part)**

python

def display\_message\_part(part):

if part.part\_kind == 'user-prompt':

with st.chat\_message("user"):

st.markdown(part.content)

elif part.part\_kind == 'text':

with st.chat\_message("assistant"):

st.markdown(part.content)

* + **Purpose**: Render different message types with appropriate styling
  + **Supported Types**: User prompts, assistant text, tool calls, tool returns
  + **UI Pattern**: Chat message containers with role-based styling

1. **run\_agent\_with\_streaming(user\_input)**

python

async def run\_agent\_with\_streaming(user\_input):

async with agent.run\_stream(

user\_input,

deps=st.session\_state.agent\_deps,

message\_history=st.session\_state.messages

) as result:

async for message in result.stream\_text(delta=True):

yield message

st.session\_state.messages.extend(result.new\_messages())

* + **Purpose**: Execute RAG agent with real-time streaming
  + **Features**: Delta streaming, message history integration, session state updates
  + **Performance**: Async generator for non-blocking UI updates

1. **main() - Core UI Function**
   * **Session Initialization**: Message history and agent dependencies
   * **UI Rendering**: Title, message display, chat input
   * **Streaming Integration**: Real-time response generation with visual feedback

**Session State Management**

**State Variables:**

python

st.session\_state.messages = [] # Conversation history

st.session\_state.agent\_deps = await get\_agent\_deps() # RAG dependencies

**Message Structure:**

* **ModelRequest**: User inputs and system prompts
* **ModelResponse**: Agent responses and tool calls
* **Part Types**: user-prompt, text, tool-call, tool-return

**Streaming Architecture**

**Implementation Pattern:**

python

# Real-time streaming with visual feedback

message\_placeholder = st.empty()

full\_response = ""

async for message in generator:

full\_response += message

message\_placeholder.markdown(full\_response + "▌")

# Final render without cursor

message\_placeholder.markdown(full\_response)

**Features:**

* **Delta Streaming**: Incremental text updates
* **Visual Feedback**: Typing cursor indicator ("▌")
* **Placeholder Management**: Single element updates for performance

**Dependencies and Integration**

**Core Dependencies:**

* streamlit: Web interface framework
* pydantic\_ai: Agent framework and message types
* asyncio: Asynchronous operation support
* dotenv: Environment configuration

**Integration Points:**

* rag\_agent.py: RAG agent instance and dependencies
* utils.py: ChromaDB client utilities
* Environment variables: OpenAI API keys, model configuration

**3. UI Architecture Diagrams**

**Runtime Architecture - UI Components**

mermaid

graph TB

subgraph "Browser Client"

USER[User Interface]

CHAT[Chat Input Widget]

MSGS[Message Display Area]

STREAM[Streaming Response Area]

end

subgraph "Streamlit Server"

APP[streamlit\_app.py]

SESSION[Session State Manager]

RENDERER[Message Renderer]

STREAMER[Streaming Handler]

end

subgraph "Backend Integration"

RAG\_AGENT[rag\_agent.py]

DEPS[RAGDeps Manager]

UTILS[utils.py]

CHROMA[ChromaDB Client]

end

subgraph "External Services"

OPENAI[OpenAI API]

DB[(ChromaDB Vector Store)]

end

%% User Interactions

USER -->|Question Input| CHAT

CHAT -->|Submit| APP

MSGS <-->|Display| RENDERER

STREAM <-->|Updates| STREAMER

%% Application Flow

APP -->|Initialize| SESSION

APP -->|Render Messages| RENDERER

APP -->|Stream Response| STREAMER

SESSION -->|Get Dependencies| DEPS

RENDERER -->|Format Messages| RENDERER

STREAMER -->|Agent Stream| RAG\_AGENT

%% Backend Integration

DEPS -->|DB Client| UTILS

RAG\_AGENT -->|Vector Search| CHROMA

RAG\_AGENT -->|LLM Calls| OPENAI

UTILS -->|Query| DB

CHROMA -->|Results| RAG\_AGENT

%% Styling

classDef frontend fill:#e1f5fe

classDef streamlit fill:#f3e5f5

classDef backend fill:#e8f5e8

classDef external fill:#fce4ec

class USER,CHAT,MSGS,STREAM frontend

class APP,SESSION,RENDERER,STREAMER streamlit

class RAG\_AGENT,DEPS,UTILS,CHROMA backend

class OPENAI,DB external

**Data Flow - UI Interaction Pipeline**

mermaid

flowchart TD

subgraph "USER INTERACTION"

START([User Accesses UI]) --> INIT[Initialize Streamlit App]

INIT --> LOAD\_STATE{Session State Exists?}

LOAD\_STATE -->|No| CREATE\_STATE[Create New Session<br/>messages = []<br/>agent\_deps = RAGDeps()]

LOAD\_STATE -->|Yes| RESTORE\_STATE[Restore Session State<br/>Load message history]

CREATE\_STATE --> RENDER\_UI[Render Chat Interface]

RESTORE\_STATE --> RENDER\_UI

RENDER\_UI --> DISPLAY\_HISTORY[Display Message History<br/>For each message part]

DISPLAY\_HISTORY --> WAIT\_INPUT[Wait for User Input]

end

subgraph "MESSAGE PROCESSING"

WAIT\_INPUT --> INPUT\_RECEIVED[User Submits Question]

INPUT\_RECEIVED --> DISPLAY\_USER[Display User Message<br/>with st.chat\_message("user")]

DISPLAY\_USER --> STREAM\_SETUP[Setup Streaming Response<br/>Create placeholder + cursor]

STREAM\_SETUP --> AGENT\_CALL[Call RAG Agent<br/>agent.run\_stream()]

end

subgraph "STREAMING RESPONSE"

AGENT\_CALL --> STREAM\_START[Start Async Stream<br/>async for message in result]

STREAM\_START --> DELTA\_UPDATE[Update UI with Delta<br/>full\_response += message]

DELTA\_UPDATE --> RENDER\_CURSOR[Render with Cursor<br/>markdown(response + "▌")]

RENDER\_CURSOR --> MORE\_DELTAS{More Deltas?}

MORE\_DELTAS -->|Yes| DELTA\_UPDATE

MORE\_DELTAS -->|No| FINAL\_RENDER[Final Render<br/>Remove cursor]

FINAL\_RENDER --> UPDATE\_HISTORY[Update Session State<br/>messages.extend(new\_messages)]

end

subgraph "STATE MANAGEMENT"

UPDATE\_HISTORY --> PRESERVE\_CONTEXT[Preserve Conversation Context<br/>For next interaction]

PRESERVE\_CONTEXT --> WAIT\_INPUT

end

%% Data Types

INPUT\_RECEIVED -.->|String| DISPLAY\_USER

STREAM\_START -.->|AsyncGenerator[str]| DELTA\_UPDATE

DELTA\_UPDATE -.->|Incremental Text| RENDER\_CURSOR

UPDATE\_HISTORY -.->|ModelMessage[]| PRESERVE\_CONTEXT

%% Styling

classDef interaction fill:#e1f5fe

classDef processing fill:#fff3e0

classDef streaming fill:#e8f5e8

classDef state fill:#fce4ec

class START,INIT,LOAD\_STATE,CREATE\_STATE,RESTORE\_STATE,RENDER\_UI,DISPLAY\_HISTORY,WAIT\_INPUT interaction

class INPUT\_RECEIVED,DISPLAY\_USER,STREAM\_SETUP,AGENT\_CALL processing

class STREAM\_START,DELTA\_UPDATE,RENDER\_CURSOR,MORE\_DELTAS,FINAL\_RENDER streaming

class UPDATE\_HISTORY,PRESERVE\_CONTEXT state

**Sequence Diagram - Complete UI Interaction Flow**

mermaid

sequenceDiagram

participant User

participant Browser

participant Streamlit as Streamlit Server

participant SessionState as Session State

participant Renderer as Message Renderer

participant Streamer as Stream Handler

participant RAGAgent as RAG Agent

participant ChromaDB

participant OpenAI as OpenAI API

%% Application Initialization

rect rgb(225, 245, 254)

Note over User,OpenAI: UI INITIALIZATION PHASE

User->>Browser: Access Streamlit App

Browser->>Streamlit: GET /

Streamlit->>Streamlit: Initialize streamlit\_app.py

Streamlit->>SessionState: Check session\_state.messages

alt New Session

SessionState->>SessionState: Initialize empty messages[]

SessionState->>SessionState: Create agent\_deps with RAGDeps

else Existing Session

SessionState->>SessionState: Load existing messages[]

SessionState->>SessionState: Restore agent\_deps

end

Streamlit->>Renderer: Display existing messages

loop For each message in history

Renderer->>Renderer: display\_message\_part(part)

Renderer->>Browser: Render chat message

end

Streamlit->>Browser: Display chat interface

Browser-->>User: Show UI with chat input

end

%% User Interaction Phase

rect rgb(232, 245, 232)

Note over User,OpenAI: QUESTION & RESPONSE PHASE

User->>Browser: Type question + Enter

Browser->>Streamlit: Submit user\_input

Streamlit->>Renderer: Display user message

Renderer->>Browser: Show user chat bubble

Streamlit->>Streamer: run\_agent\_with\_streaming(user\_input)

Streamer->>RAGAgent: agent.run\_stream(input, deps, history)

RAGAgent->>RAGAgent: Analyze question

RAGAgent->>ChromaDB: Vector similarity search

ChromaDB-->>RAGAgent: Relevant document chunks

RAGAgent->>OpenAI: Chat completion with context

OpenAI-->>RAGAgent: Start streaming response

loop Streaming Response

RAGAgent-->>Streamer: yield message\_delta

Streamer->>Streamer: full\_response += delta

Streamer->>Browser: Update placeholder with cursor

Browser-->>User: Show partial response + "▌"

end

RAGAgent-->>Streamer: Stream complete

Streamer->>Browser: Final render without cursor

Streamer->>SessionState: Update messages with new\_messages()

Browser-->>User: Show complete response

end

%% Continuous Interaction

rect rgb(255, 243, 224)

Note over User,SessionState: CONVERSATION CONTINUATION

User->>Browser: Ask follow-up question

Note over Browser,SessionState: Repeat interaction flow with preserved context

SessionState->>RAGAgent: Include full message history

RAGAgent->>OpenAI: Context-aware response with conversation history

end

**UI Component State Diagram**

mermaid

stateDiagram-v2

[\*] --> AppStartup

AppStartup --> SessionCheck : Initialize Streamlit

state SessionCheck {

[\*] --> CheckState

CheckState --> NewSession : No existing state

CheckState --> ExistingSession : Has message history

NewSession --> InitializeState : Create empty messages[]

ExistingSession --> RestoreState : Load message history

InitializeState --> [\*] : Ready

RestoreState --> [\*] : Ready

}

SessionCheck --> UIReady : Session Prepared

state UIReady {

[\*] --> DisplayHistory

DisplayHistory --> WaitingForInput : Render complete

WaitingForInput --> ProcessingInput : User submits question

state ProcessingInput {

[\*] --> DisplayUserMessage

DisplayUserMessage --> InitializeStreaming

InitializeStreaming --> StreamingResponse

state StreamingResponse {

[\*] --> CreatePlaceholder

CreatePlaceholder --> StartStream

StartStream --> UpdateDelta : Receive text chunk

UpdateDelta --> RenderWithCursor : Update UI

RenderWithCursor --> UpdateDelta : More chunks

RenderWithCursor --> FinalRender : Stream complete

FinalRender --> [\*] : Response complete

}

StreamingResponse --> UpdateSessionState

UpdateSessionState --> [\*] : Ready for next input

}

ProcessingInput --> WaitingForInput : Response complete

}

state ErrorHandling {

[\*] --> DetectError

DetectError --> AgentError : RAG agent failure

DetectError --> StreamingError : Streaming failure

DetectError --> UIError : UI rendering failure

AgentError --> DisplayErrorMessage : Show error to user

StreamingError --> ResetStream : Restart streaming

UIError --> RefreshInterface : Reload UI

DisplayErrorMessage --> [\*] : Error handled

ResetStream --> [\*] : Streaming restored

RefreshInterface --> [\*] : UI restored

}

UIReady --> ErrorHandling : Error detected

ErrorHandling --> UIReady : Error resolved

UIReady --> AppShutdown : User closes browser

AppShutdown --> [\*]

**Message Flow and Data Structure**

mermaid

flowchart LR

subgraph "INPUT PROCESSING"

USER\_INPUT[User Question<br/>String]

CHAT\_WIDGET[st.chat\_input()<br/>Streamlit Widget]

end

subgraph "SESSION STATE"

MESSAGES[st.session\_state.messages<br/>List[ModelMessage]]

AGENT\_DEPS[st.session\_state.agent\_deps<br/>RAGDeps Object]

end

subgraph "MESSAGE TYPES"

MODEL\_REQ[ModelRequest<br/>{parts: [UserPromptPart]}]

MODEL\_RESP[ModelResponse<br/>{parts: [TextPart, ToolCallPart]}]

PART\_TYPES[Message Parts<br/>• user-prompt<br/>• text<br/>• tool-call<br/>• tool-return]

end

subgraph "STREAMING DATA"

STREAM\_GEN[AsyncGenerator[str]<br/>Delta text chunks]

FULL\_RESP[Accumulated Response<br/>String]

PLACEHOLDER[st.empty()<br/>UI Placeholder]

end

subgraph "RENDERING OUTPUT"

CHAT\_MSG[st.chat\_message()<br/>Chat Container]

MARKDOWN[st.markdown()<br/>Formatted Content]

CURSOR[Typing Indicator<br/>"response + ▌"]

end

%% Data Flow

USER\_INPUT --> CHAT\_WIDGET

CHAT\_WIDGET --> MODEL\_REQ

MODEL\_REQ --> MESSAGES

MESSAGES --> STREAM\_GEN

AGENT\_DEPS --> STREAM\_GEN

STREAM\_GEN --> FULL\_RESP

FULL\_RESP --> PLACEHOLDER

PLACEHOLDER --> CURSOR

CURSOR --> MARKDOWN

MODEL\_RESP --> MESSAGES

PART\_TYPES --> CHAT\_MSG

CHAT\_MSG --> MARKDOWN

%% Data Transformations

USER\_INPUT -.->|"user\_input"| MODEL\_REQ

STREAM\_GEN -.->|"async for delta"| FULL\_RESP

FULL\_RESP -.->|"response + '▌'"| CURSOR

MODEL\_RESP -.->|"part.content"| MARKDOWN

%% Styling

classDef input fill:#e1f5fe

classDef state fill:#f3e5f5

classDef types fill:#e8f5e8

classDef streaming fill:#fff3e0

classDef output fill:#fce4ec

class USER\_INPUT,CHAT\_WIDGET input

class MESSAGES,AGENT\_DEPS state

class MODEL\_REQ,MODEL\_RESP,PART\_TYPES types

class STREAM\_GEN,FULL\_RESP,PLACEHOLDER streaming

class CHAT\_MSG,MARKDOWN,CURSOR output

**Key Technical Innovations**

**Streaming Integration**

* **Real-time Updates**: Delta-based streaming with visual feedback
* **Async Generators**: Non-blocking UI updates during LLM generation
* **Placeholder Management**: Efficient DOM updates for streaming text

**Session State Management**

* **Conversation Persistence**: Full message history preservation
* **Context Awareness**: Multi-turn conversation support
* **Dependency Injection**: Clean separation of UI and business logic

**Message Part Rendering**

* **Flexible Display**: Different rendering for user prompts, tool calls, responses
* **Extensible Design**: Easy addition of new message part types
* **Clean UI Patterns**: Consistent chat interface styling

This Streamlit UI provides a production-ready interface for Project Citadel's RAG capabilities, with enterprise-grade streaming, session management, and user experience features.

Compare with GPT-4.1