WhatsApp ↔ Geneline-X AI Bridge System

Comprehensive System Architecture Presentation

📋 Table of Contents

System Overview

What is the WhatsApp-Geneline Bridge?

The WhatsApp-Geneline Bridge is a sophisticated intermediary service that creates a seamless connection between WhatsApp messaging and the Geneline-X artificial intelligence platform. Think of it as a highly intelligent translator that sits between users on WhatsApp and a powerful AI system.

The Core Purpose: This system was specifically designed to provide public health information to users in Sierra Leone through WhatsApp. Since WhatsApp is one of the most widely used communication platforms in Sierra Leone, making health information accessible through this channel dramatically increases reach and impact.

Why This System Exists: Traditional chatbots are simple question-and-answer systems. This bridge service goes far beyond that by implementing an intelligent agent framework that can:

1. **Understand context** - It remembers previous messages in a conversation, so users can ask follow-up questions naturally
2. **Access multiple knowledge sources** - It can search through uploaded medical documents, query knowledge bases, and even search the internet when needed
3. **Reason through complex questions** - Using the agent framework, it can break down complex health questions and gather information from multiple sources before responding
4. **Stay focused** - With configurable system prompts, it's designed to stay on topic (public health) and politely redirect off-topic conversations

Key Capabilities

```
graph LR
    A[WhatsApp Users] -->|Messages| B[Bridge Service]
    B -->|AI Queries| C[Geneline-X AI]
    B -->|Document Upload| D[Knowledge Base]
    B -->|Web Search| E[Internet]
    C -->|Responses| B
    B -->|Replies| A

    style B fill:#4CAF50,stroke:#2E7D32,color:#fff
```

```
    style C fill:#2196F3,stroke:#1565C0,color:#fff
    style D fill:#FF9800,stroke:#E65100,color:#fff
```

Detailed Explanation of System Capabilities

1. Answer Questions About Public Health The system uses a system prompt that instructs the AI to focus exclusively on public health topics related to Sierra Leone. This includes:

- Disease information (symptoms, treatment, prevention)
- Vaccination schedules and locations
- Healthcare facility information
- Maternal and child health
- Nutrition and sanitation
- Hygiene practices
- Disease prevention strategies

2. Search Knowledge Bases When you upload documents (like the Standard Treatment Guidelines for Sierra Leone), the system can intelligently search through them. When a user asks a specific question that would benefit from document knowledge, the agent automatically triggers a knowledge search tool, retrieves relevant information, and incorporates it into the response.

3. Process Documents to Expand Knowledge Healthcare administrators can upload PDF documents through either a command-line interface or REST API. The system sends these to Geneline-X for processing, which extracts the text, indexes it, and makes it searchable. This means the chatbot's knowledge can be continuously expanded without touching any source files.

4. Maintain Conversation Context Unlike simple chatbots that treat each message independently, this system maintains a conversation history for each chat. It remembers the last 10 messages, which allows users to ask follow-up questions like "What about for children?" after asking about malaria treatment, and the system understands the context.

5. Access the Internet For questions that require current information (like disease outbreak statistics), the agent can use a web search tool to find up-to-date information from the internet, then synthesize that information into a helpful response.

Core Architecture

High-Level System Components

The WhatsApp-Geneline Bridge is built using a modular architecture where each component has a specific responsibility. This design makes the system maintainable, testable, and scalable. Let's understand how these components work together.

Architecture Overview: The system is divided into four logical layers:

1. **WhatsApp Layer** - Handles all communication with WhatsApp servers
2. **Processing Layer** - Manages queuing, rate limiting, and message orchestration
3. **AI Layer** - Contains the intelligent agent and tools for enhanced capabilities
4. **Infrastructure** - Provides configuration, logging, and admin control

The system consists of **9 major components** working together:

```
graph TB
    subgraph "WhatsApp Layer"
        WC[WhatsApp Client]
        MH[Message Handler]
    end

    subgraph "Processing Layer"
        QM[Queue Manager]
        MW[Message Worker]
        AL[Agent Loop]
    end

    subgraph "AI Layer"
        GC[Geneline Client]
        TR[Tool Registry]
        KS[Knowledge Search]
        WS[Web Search]
    end

    subgraph "Infrastructure"
        API[API Server]
        LOG[Logger]
        CFG[Configuration]
    end

    WC -->|Messages| MH
    MH -->|Enqueue| QM
    QM -->|Process| MW
    MW -->|Execute| AL
    AL -->|Tools| TR
    TR --> KS
    TR --> WS
    AL -->|AI Request| GC
    GC -->|Response| AL
    AL -->|Send| MW
    MW -->|Reply| MH
    MH -->|Send| WC

    API -.->|Manage| QM
    API -.->|Monitor| WC

    style WC fill:#25D366,stroke:#128C7E,color:#fff
    style GC fill:#2196F3,stroke:#1565C0,color:#fff
    style AL fill:#FF9800,stroke:#E65100,color:#fff
```

Component Responsibilities

Component	Responsibility
WhatsApp Client	Connects to WhatsApp, authenticates, receives/sends messages

Message Handler	Filters messages, downloads media, routes to queue
Queue Manager	FIFO queue with rate limiting per chat
Message Worker	Coordinates message processing and response delivery
Agent Loop	Orchestrates AI reasoning and tool usage
Geneline Client	Communicates with Geneline-X AI API
Tool Registry	Manages available tools, parses and executes tool calls
API Server	REST endpoints for admin control and monitoring
Logger	Structured logging for debugging and monitoring

How Components Interact

Step-by-Step Component Interaction:

1. **WhatsApp Client receives a message** from WhatsApp servers when a user sends a message to the bot
2. **Message Handler** processes the incoming message:
    - Checks if it's from a group (and filters if groups are disabled)
    - Downloads any media attachments (images, PDFs, audio, video)
    - Converts media to base64 encoding for transmission
    - Packages everything into a structured message object
3. **Queue Manager** receives the message and:
    - Creates a queue for this specific chat if it doesn't exist
    - Adds the message to the end of that chat's queue (FIFO - First In, First Out)
    - Tracks the queue size and processing metrics
4. **Message Worker** picks up messages from the queue:
    - Checks if there are available worker slots (max 5 concurrent)
    - Verifies rate limiting (waits if the last message from this chat was too recent)
    - Once cleared, executes the agent loop
5. **Agent Loop** is the brain of the operation:
    - Retrieves conversation history for context
    - Builds a prompt with system instructions and tools list
    - Calls Geneline-X AI through the Geneline Client
    - Parses the response to check for tool calls
    - If tools are needed, executes them through Tool Registry
    - Sends the final response back to Message Worker
6. **Message Worker** sends the response:
    - Receives the final answer from Agent Loop
    - Calls Message Handler to send it back
    - Message Handler uses WhatsApp Client to send the message
    - Updates metrics and logs the completion

Why This Architecture?

- **Separation of Concerns**: Each component does one thing well, making debugging easier
- **Scalability**: The queue system allows handling multiple conversations simultaneously
- **Reliability**: If one message fails, it doesn't affect others
- **Maintainability**: You can update the AI logic without touching WhatsApp code
- **Testability**: Each component can be tested independently

---

Message Processing Flow

Complete Message Lifecycle

Understanding the Journey of a Message:

When a user sends a message to the WhatsApp bot, it goes through a sophisticated pipeline that ensures reliable processing, intelligent responses, and proper error handling. Let's trace this journey step by step.

The Full Process Explained:

The flowchart below shows every decision point and action from the moment a message arrives until the user receives a response. Each diamond shape represents a decision (yes/no question), and each rectangle represents an action taken by the system.

```
flowchart TD
    Start([User Sends WhatsApp Message]) --> Receive[WhatsApp Client Receives
Message]
    Receive --> Filter{Is Group Message?}
    Filter -->|Yes & Groups Disabled| Ignore[Ignore Message]
    Filter -->|No| CheckMedia{Has Media?}

    CheckMedia -->|Yes| DownloadMedia[Download Media Attachment]
    CheckMedia -->|No| Enqueue
    DownloadMedia --> ConvertMedia[Convert to Base64]
    ConvertMedia --> Enqueue

    Enqueue[Add to Queue Manager] --> RateCheck{Rate Limit OK?}
    RateCheck -->|No| Wait[Wait for Rate Limit]
    Wait --> RateCheck
    RateCheck -->|Yes| WorkerPick[Message Worker Picks Up]

    WorkerPick --> AgentStart[Agent Loop Starts]
    AgentStart --> BuildPrompt[Build Prompt with History]
    BuildPrompt --> CallAI[Call Geneline-X AI]

    CallAI --> ParseResponse{Tool Call Detected?}
    ParseResponse -->|No| FinalResponse[Final Response]
    ParseResponse -->|Yes| ExecuteTool[Execute Tool]

    ExecuteTool --> ToolResult[Get Tool Result]
    ToolResult --> AddHistory[Add to Conversation History]
    AddHistory --> CallAI2[Call AI with Tool Result]
    CallAI2 --> FinalResponse
```

```
    FinalResponse --> UpdateHistory[Update Conversation History]
    UpdateHistory --> SendWhatsApp[Send Response to WhatsApp]
    SendWhatsApp --> End([User Receives Reply])

    Ignore --> End

    style Start fill:#4CAF50,stroke:#2E7D32,color:#fff
    style End fill:#4CAF50,stroke:#2E7D32,color:#fff
    style CallAI fill:#2196F3,stroke:#1565C0,color:#fff
    style ExecuteTool fill:#FF9800,stroke:#E65100,color:#fff
    style SendWhatsApp fill:#25D366,stroke:#128C7E,color:#fff
```

Detailed Breakdown of Each Step

1. Message Reception (Green Start Node) A user opens WhatsApp on their phone and sends a message to the bot's number. This could be a simple text message, a question, or even a message with an attached image or document.

2. WhatsApp Client Receives Message The WhatsApp Client (using whatsapp-web.js library) maintains a persistent connection to WhatsApp's servers. When a message arrives, it triggers an event that the system captures.

3. Group Message Filter (First Decision Point) By default, the system is configured to ignore group messages because:

- Group messages create high volume and could overwhelm the system
- Privacy concerns (the bot would respond to everyone in the group)
- Resource management (focusing on one-on-one conversations)

If `ALLOW_GROUP_MESSAGES=true` is set in configuration, this filter allows group messages through.

4. Media Detection and Processing If the message contains media (photo, document, audio, video):

- The system downloads it from WhatsApp servers
- Converts it to base64 encoding (a text representation of binary data)
- Attaches it to the message object for later processing
- This allows the AI to potentially analyze images or process document content

5. Queue Enqueueing The Message Handler sends the processed message to the Queue Manager. The queue manager:

- Creates a separate queue for each unique chat ID if one doesn't exist
- Adds this message to the end of that chat's queue
- Returns immediately (doesn't wait for processing)

6. Rate Limiting Check Before processing, the system checks:

- When was the last message from this chat processed?
- Has enough time passed? (default: 1000ms = 1 second)
- If no: wait until the time has passed
- If yes: proceed to worker

This prevents one user from flooding the system and ensures fair processing.

7. Message Worker Processing The worker:

- Checks if there are fewer than 5 active processing jobs (concurrency limit)
- If there's capacity, picks up the message
- Starts the Agent Loop to generate a response

8. Agent Loop and AI Call (Blue Node) This is where the magic happens:

- Retrieves the conversation history (last 10 messages with this user)
- Builds a comprehensive prompt including:
  - System instructions ("You are a public health assistant for Sierra Leone")
  - Conversation history
  - Available tools description
  - Current user message
- Sends request to Geneline-X AI API
- Waits for streaming response

9. Tool Call Detection (Critical Decision) The Agent Loop examines the AI's response looking for a special pattern that indicates tool usage:

- Example: `<tool>knowledge_search</tool><query>malaria treatment guidelines</query>`
- If found: this is a tool call, not a final answer
- If not found: this is the final response to send back

10. Tool Execution (Orange Node) When a tool call is detected:

- The Tool Registry validates the tool name
- Executes the appropriate tool:
  - `knowledge_search` : Queries Geneline-X knowledge base for relevant documents
  - `web_search` : Performs internet search for current information
- Returns results to the Agent Loop

11. Tool Result Processing The Agent Loop:

- Takes the tool results
- Adds them to the conversation history ("Tool: knowledge_search returned: ...")
- Calls the AI again with: "Given these search results, answer the user's question"
- This second AI call produces the final answer incorporating the tool results

12. Conversation History Update Before sending the response:

- Add the user's message to history
- Add any tool calls to history
- Add the final assistant response to history
- Trim history if it exceeds 10 messages (keeping most recent)

13. Send Response to WhatsApp (Green Node) The Message Worker:

- Receives the final response text
- Tells Message Handler to send it
- Message Handler uses WhatsApp Client
- Message appears in user's WhatsApp conversation

14. Metrics and Logging Throughout this entire process, every step is logged:

- Timestamps for performance monitoring
- Success/failure status for reliability tracking
- Tool usage statistics

- Error details if something goes wrong

Message Queue System

The queue system ensures reliable processing with concurrency control:

```
graph TB
    subgraph "Chat A Queue"
        A1[Message 1] --> A2[Message 2]
        A2 --> A3[Message 3]
    end

    subgraph "Chat B Queue"
        B1[Message 1] --> B2[Message 2]
    end

    subgraph "Chat C Queue"
        C1[Message 1] --> C2[Message 2]
        C2 --> C3[Message 3]
        C3 --> C4[Message 4]
    end

    A1 -.->|Worker 1| P1[Processing]
    B1 -.->|Worker 2| P2[Processing]
    C1 -.->|Worker 3| P3[Processing]

    P1 --> R1[Rate Limit Check]
    P2 --> R2[Rate Limit Check]
    P3 --> R3[Rate Limit Check]

    R1 --> W1[Worker Thread]
    R2 --> W2[Worker Thread]
    R3 --> W3[Worker Thread]

    style A1 fill:#4CAF50,stroke:#2E7D32,color:#fff
    style B1 fill:#4CAF50,stroke:#2E7D32,color:#fff
    style C1 fill:#4CAF50,stroke:#2E7D32,color:#fff
```

Understanding the Queue System

Why Separate Queues Per Chat?

Imagine three users (Chat A, B, and C) are all using the bot simultaneously:

- User A sends 3 messages rapidly
- User B sends 2 messages
- User C sends 4 messages

Without separate queues, these messages would mix together and User A's second message might be processed before User A's first message gets a response. This would break conversation context.

The Solution: Per-Chat FIFO Queues

- Each WhatsApp chat gets its own queue

- Messages from Chat A are processed in order (First In, First Out)
- Messages from Chat B are processed separately, in their own order
- The queues are independent, so Chat A's processing doesn't block Chat B

How Workers Pick Up Messages:

1. The Queue Manager constantly monitors all queues
2. When a worker is free, it looks across all chat queues
3. For each queue, it checks:
    - Is there a message waiting?
    - Has the rate limit cooldown passed for this chat?
    - If yes to both: assign to this worker
4. Worker becomes busy processing that message
5. When done, worker becomes free and picks up the next eligible message

Concurrency Control:

The system allows up to 5 messages to be processed simultaneously (configurable via `MAX_CONCURRENCY`). This means:

- Worker 1 might be processing Chat A's message
- Worker 2 might be processing Chat B's message
- Worker 3 might be processing Chat C's message
- Workers 4 and 5 are ready for more messages

Why limit to 5? Because each AI request:

- Consumes memory
- Makes API calls to Geneline-X (which has its own rate limits)
- Prevents overwhelming the system

Rate Limiting Per Chat:

Each chat has a minimum time between message processing (default: 1000ms). This means:

- If Chat A's message is processed at 10:00:00
- And another message from Chat A arrives at 10:00:00.500 (half a second later)
- The system will wait 500ms more before processing it
- This prevents any single user from monopolizing resources

Retry Logic with Exponential Backoff:

If the AI API returns an error:

- **429 Too Many Requests**: Wait 2 seconds, then 4, then 8 (exponential backoff)
- **5xx Server Error**: Same exponential backoff
- **4xx Client Error** (except 429): Don't retry, return error to user

This pattern prevents hammering a failing service and gives it time to recover.

Queue Features:

- **Per-Chat FIFO**: Messages from the same chat are processed in order
- **Concurrency Control**: Max 5 concurrent AI requests (configurable)
- **Rate Limiting**: Minimum 1000ms between messages per chat (configurable)
- **Retry Logic**: Exponential backoff for 429/5xx errors

Agent Framework

Intelligent Conversation Orchestration

What is the Agent Framework?

The Agent Framework is the most sophisticated part of this system. It transforms Geneline-X from a simple question-answer AI into an intelligent agent that can:

- **Think multi-step**: "I need to search the knowledge base first, then answer"
- **Use tools**: Access knowledge bases, search the web, potentially more in the future
- **Maintain context**: Remember the conversation and reference previous messages
- **Retry intelligently**: Handle errors and rate limits gracefully

Why an Agent (vs. Simple Chatbot)?

A traditional chatbot workflow:

1. User asks: "What does the treatment guideline say about malaria?"
2. AI responds with general knowledge
3. Done

An agent workflow:

1. User asks: "What does the treatment guideline say about malaria?"
2. Agent thinks: "This requires specific document knowledge"
3. Agent calls `knowledge_search` tool with query "malaria treatment guidelines"
4. Tool returns: "Section 5.2: Malaria treatment protocol involves..."
5. Agent thinks: "Now I have the specific information"
6. Agent responds: "According to the Standard Treatment Guidelines for Sierra Leone, Section 5.2..."
7. Done

The agent can reason about what tools it needs and when to use them.

The Agent Loop is the brain of the system, enabling multi-step reasoning:

```
flowchart TD
    Start([Agent Loop Starts]) --> AddUser[Add User Message to History]
    AddUser --> Iteration{Iteration < Max?}

    Iteration -->|No| MaxReached[Return Fallback Response]
    Iteration -->|Yes| History[Get Conversation History]

    History --> BuildPrompt[Build Prompt with Tools List]
    BuildPrompt --> CallGeneline[Call Geneline-X API]

    CallGeneline --> Success{Request Successful?}
    Success -->|No| Retry{Retries Left?}
    Retry -->|Yes| Backoff[Exponential Backoff]
    Backoff --> CallGeneline
    Retry -->|No| Error[Return Error Response]

    Success -->|Yes| ParseTool{Tool Call in Response?}
```

```
    ParseTool -->|No| IsFinal[This is Final Response]
    IsFinal --> AddAssistant[Add to History]
    AddAssistant --> ReturnResponse([Return Response to User])

    ParseTool -->|Yes| ValidTool{Valid Tool?}
    ValidTool -->|No| IsFinal
    ValidTool -->|Yes| LogTool[Log Tool Call]

    LogTool --> Execute[Execute Tool]
    Execute --> CheckTool{Which Tool?}

    CheckTool -->|Knowledge Search| KS[Query Knowledge Base]
    CheckTool -->|Web Search| WS[Search Internet]

    KS --> ToolResult[Get Tool Result]
    WS --> ToolResult

    ToolResult --> AddToolHistory[Add Tool Result to History]
    AddToolHistory --> BuildPrompt2[Build Prompt with Tool Result]
    BuildPrompt2 --> CallGeneline2[Call Geneline-X Again]

    CallGeneline2 --> FinalResp[Get Final Response]
    FinalResp --> AddAssistant

    MaxReached --> ReturnResponse
    Error --> ReturnResponse

    style Start fill:#4CAF50,stroke:#2E7D32,color:#fff
    style CallGeneline fill:#2196F3,stroke:#1565C0,color:#fff
    style Execute fill:#FF9800,stroke:#E65100,color:#fff
    style ReturnResponse fill:#4CAF50,stroke:#2E7D32,color:#fff
```

Complete Breakdown of the Agent Loop

Initialization Phase:

1. **Agent Loop Starts** - The Message Worker calls the Agent Loop's `run()` method with:
    - `chatId`: Unique identifier for this WhatsApp conversation
    - `userMessage`: The text the user sent
2. **Add User Message to History** - Before doing anything else, the system records this message in the conversation history with:
    - Role: "user"
    - Content: the message text
    - Timestamp: current time

Iteration Control:

3. **Check Iteration Count** - The agent can loop up to 5 times (configurable). Why?
    - Prevents infinite loops if something goes wrong
    - Each iteration typically involves calling the AI at least once

- More iterations = more API calls = more cost and time

If max iterations are reached without a final answer, return a fallback message: "I'm having trouble processing your request."

Building the AI Request:

4. **Get Conversation History** - Retrieve the last 10 messages from this chat:

   ```
   User: What are malaria symptoms?
   Assistant: Malaria symptoms include fever, chills...
   User: What about for children?
   ```

   This provides context so the AI understands "for children" refers to malaria.

5. **Build Prompt with Tools List** - The prompt builder creates a comprehensive prompt:

   ```
   System: You are a public health assistant for Sierra Leone.

   You have access to these tools:
   - knowledge_search: Search the knowledge base for specific information
   - web_search: Search the internet for current information

   To use a tool, respond with:
   <tool>tool_name</tool>
   <query>your search query</query>
   <thought>why you're using this tool</thought>

   Conversation History:
   User: What are malaria symptoms?
   Assistant: Malaria symptoms include...
   User: What about for children?

   User's Current Message: What about for children?
   ```

Calling the AI:

6. **Call Geneline-X API** - Send the complete prompt to the AI
   - The request includes the chatbot ID and API key
   - Response is streamed back token by token
   - System buffers it into complete text
7. **Request Success Check** - Did the API call work?
   - **If No**: Check if we have retries left
   - **If retries left**: Wait (exponential backoff: 1s, 2s, 4s, 8s)
   - **If no retries**: Return error message to user
   - **If Yes**: Proceed to parse the response

Critical Decision: Is This a Tool Call?

8. **Parse Tool Call** - The Tool Registry examines the AI's response looking for the pattern:

```
<tool>knowledge_search</tool>
<query>malaria treatment guidelines children</query>
<thought>User is asking about children specifically, need to check treatment
guidelines</thought>
```

9. **Tool Call Detected?**
    - **If No**: This is a regular text response, this is the final answer!
    - **If Yes**: This is a request to use a tool, need to execute it

When It's a Final Response (No Tool):

10. **Add to History** - Store the assistant's response in conversation history
11. **Return Response** - Send this response back to the user via WhatsApp
12. **Done!**

When It's a Tool Call:

13. **Validate Tool** - Is "knowledge_search" a registered tool? YES
14. **Log Tool Call** - Record for monitoring:
    - Tool name: knowledge_search
    - Query: malaria treatment guidelines children
    - Thought: User asking about children
15. **Execute Tool** - Different tools do different things:

Knowledge Search Tool:

```
Input: "malaria treatment guidelines children"
Process:
- Calls Geneline-X knowledge API with query
- Searches through all uploaded documents
- Returns relevant excerpts
Output: "Found in Standard Treatment Guidelines:
         Section 5.2.1 - Children under 5 with severe malaria..."
```

Web Search Tool:

```
Input: "current malaria cases sierra leone 2024"
Process:
- Calls web search API
- Gets current web results
- Extracts relevant information
Output: "According to WHO website (2024): Sierra Leone reported..."
```

16. **Get Tool Result** - The tool returns its findings
17. **Add Tool Result to History** - Store in conversation:

```
Role: tool
Tool Name: knowledge_search
Content: "Found in Standard Treatment Guidelines: Section 5.2.1..."
```

18. **Build Prompt with Tool Result** - Create a new prompt:

```
Previous conversation...

You used the knowledge_search tool and it returned:
"Found in Standard Treatment Guidelines: Section 5.2.1 -
 Children under 5 with severe malaria should receive..."

Now answer the user's question using this information.
User's question was: What about for children?
```

19. **Call Geneline-X Again** - Send this new prompt to AI

20. **Get Final Response** - AI responds:

```
For children with malaria, according to the Standard Treatment
Guidelines for Sierra Leone, Section 5.2.1 states that children
under 5 with severe malaria should receive [specific treatment]...
```

21. **Add Final Response to History**

22. **Return Response to User**

Why This Design?

- **Accuracy**: By using tools, answers can reference actual documents rather than relying on AI memory
- **Currency**: Web search tool provides up-to-date information
- **Transparency**: Tool usage is logged, so you can see when the bot searches vs. uses general knowledge
- **Extensibility**: Easy to add new tools (e.g., appointment booking, facility locator)

Tool System Architecture

The agent can use tools to enhance its capabilities:

```
graph TB
    subgraph "Agent Loop"
        AL[Agent Loop]
    end

    subgraph "Tool Registry"
        TR[Tool Registry]
        Parse[Parse Tool Call]
        Validate[Validate Tool]
        Execute[Execute Tool]
    end

    subgraph "Available Tools"
        KT[Knowledge Search Tool]
        WT[Web Search Tool]
    end
```

```
    AL -->|Response with Tool Call| Parse
    Parse -->|Extract Tool Name & Args| Validate
    Validate -->|Check if Registered| Execute

    Execute -->|knowledge_search| KT
    Execute -->|web_search| WT

    KT -->|Query Geneline-X KB| KB[Knowledge Base API]
    WT -->|Query Search API| WEB[Web Search API]

    KB -->|Results| Result[Tool Result]
    WEB -->|Results| Result

    Result -->|Return to| AL

    style AL fill:#FF9800,stroke:#E65100,color:#fff
    style TR fill:#9C27B0,stroke:#6A1B9A,color:#fff
    style KT fill:#2196F3,stroke:#1565C0,color:#fff
    style WT fill:#00BCD4,stroke:#00838F,color:#fff
```

How the Tool System Works

Tool Registration:

When the system starts, each tool registers itself with the Tool Registry:

1. **Knowledge Search Tool** registers:
    - Name: "knowledge_search"
    - Description: "Search the knowledge base for specific health information"
    - Parameters: query (string)
    - Execute function: Calls Geneline-X knowledge API
2. **Web Search Tool** registers:
    - Name: "web_search"
    - Description: "Search the internet for current information"
    - Parameters: query (string)
    - Execute function: Calls web search API

Tool Call Parsing:

The Tool Registry has a parser that examines AI responses looking for this XML-like pattern:

```
<tool>tool_name</tool>
<query>search query here</query>
<thought>reasoning for using this tool</thought>
```

If found, it extracts:

- Tool name
- Query/arguments
- The AI's reasoning (logged for monitoring)

Tool Validation:

Before executing, the Tool Registry checks:

1. Is this tool name registered?
2. Are the required parameters provided?
3. Is the user allowed to use this tool?

If validation fails, return an error message instead of executing.

Tool Execution:

Each tool has an `execute()` method:

Knowledge Search Tool Execute:

```
Async execute(query: string) {
  1. Call Geneline-X knowledge API
  2. POST /knowledge/search with query
  3. Receive relevant document excerpts
  4. Format results as text
  5. Return to Agent Loop
}
```

Web Search Tool Execute:

```
Async execute(query: string) {
  1. Call web search API
  2. Get top 5 results
  3. Extract titles and snippets
  4. Format as readable text
  5. Return to Agent Loop
}
```

Adding New Tools:

The system is designed for easy tool expansion. To add a new tool:

1. Create file: `src/agent/tools/new-tool.ts`
2. Implement the Tool interface:

```
class NewTool implements Tool {
  name = "new_tool";
  description = "What this tool does";

  async execute(parameters) {
    // Implementation
    return result;
  }
}
```

3. Register in `tool-registry.ts`
4. Done! Agent can now use it

Examples of future tools:

- **facility_locator**: Find nearby health facilities
- **appointment_scheduler**: Book health appointments
- **symptom_checker**: Interactive symptom assessment
- **translation_tool**: Translate to local languages

Conversation History Management

```
graph LR
    subgraph "Conversation History"
        H1[Message 1: User] --> H2[Message 2: Assistant]
        H2 --> H3[Message 3: User]
        H3 --> H4[Message 4: Tool Call]
        H4 --> H5[Message 5: Tool Result]
        H5 --> H6[Message 6: Assistant]
        H6 --> H7[Message 7: User]
    end

    H7 -.->|Max 10 Messages| Limit[History Limit]
    Limit -.->|Truncate Oldest| H1

    style H1 fill:#4CAF50,stroke:#2E7D32,color:#fff
    style H4 fill:#FF9800,stroke:#E65100,color:#fff
    style H7 fill:#4CAF50,stroke:#2E7D32,color:#fff
```

Understanding Conversation Memory

Why Conversation History Matters:

Without memory:

```
User: What are malaria symptoms?
Bot: Fever, chills, headache...

User: What about treatment?
Bot: Treatment for what?
```

With memory:

```
User: What are malaria symptoms?
Bot: Fever, chills, headache...

User: What about treatment?
Bot: For malaria treatment, you should... (bot remembers context)
```

How It Works:

The Conversation History manager maintains a separate history for each chat ID:

```
Chat ID: 1234567890@c.us
History:
  1. {role: "user", content: "What are malaria symptoms?", timestamp: ...}
  2. {role: "assistant", content: "Malaria symptoms include...", timestamp: ...}
```

```
3. {role: "user", content: "What about for children?", timestamp: ...}
4. {role: "assistant", content: "For children, symptoms may...", timestamp: ...}
```

Message Roles:

- **user**: Messages from the WhatsApp user
- **assistant**: Responses from the AI
- **tool**: Tool call requests from the AI
- **tool** (result): Results returned by tool execution

History Limits:

The system keeps the **last 10 messages** per chat. Why 10?

- **Too few** (e.g., 3): Can't maintain context for complex multi-turn conversations
- **Too many** (e.g., 100):
  - Large prompts = slower AI responses
  - Large prompts = higher API costs
  - Information overload = AI may get confused
- **10 is the sweet spot**: Enough for meaningful context, small enough for efficiency

Automatic Truncation:

When the 11th message arrives:

1. System removes the oldest message (message #1)
2. Keeps messages #2-#11
3. This "sliding window" maintains recent context

Formatting for AI:

When calling the AI, history is formatted as:

```
Conversation History:
User: What are malaria symptoms?
Assistant: Malaria symptoms include fever, chills, headache, nausea...
User: What about for children?

Current Question: [latest user message]
```

This gives the AI full context to understand follow-up questions.

Memory Storage:

Currently, history is stored **in-memory** (RAM):

- **Advantage**: Very fast access
- **Disadvantage**: Lost when system restarts

For production at scale, this could be moved to:

- Redis (fast, persistent cache)
- Database (permanent storage)
- File system (simple persistence)

Features:

- Maintains last 10 messages per conversation (configurable)
- Includes user messages, assistant responses, tool calls, and tool results
- Formatted as context for AI prompts
- Enables contextual follow-up questions

Document Ingestion System

Knowledge Base Training Flow

What is Document Ingestion?

Document ingestion is the process of feeding knowledge into the chatbot. Think of it like teaching a student by giving them textbooks to study. When you upload a PDF (like the Standard Treatment Guidelines for Sierra Leone), the system:

1. Sends it to Geneline-X
2. Geneline-X extracts the text
3. Breaks it into searchable chunks
4. Indexes it for fast retrieval
5. Makes it available to the knowledge_search tool

Why This Matters:

Without document ingestion:

- Bot relies only on what the AI was trained on (general knowledge)
- May have outdated or incorrect information
- Cannot cite specific local guidelines

With document ingestion:

- Bot can reference official treatment guidelines
- Provides accurate, authoritative answers
- Can cite specific sections and page numbers
- Knowledge grows over time as you add more documents

Two Ways to Upload Documents:

1. **CLI Tool** (Command Line Interface):
    - Run a command from your terminal
    - Best for: Human administrators, one-off uploads, testing
    - Example: `npm run ingest file ./treatment-guide.pdf`
2. **Admin API** (REST API endpoints):
    - Send HTTP requests programmatically
    - Best for: Automated systems, bulk uploads, integrations
    - Example: `POST /admin/ingest/file` with file attachment

The Complete Ingestion Process:

```
flowchart TD
    Start([Upload Document]) --> Source{Upload Method?}
```

```
    Source -->|CLI Tool| CLI[Command Line]
    Source -->|API Endpoint| API[Admin API]

    CLI --> Parse[Parse Command Arguments]
    API --> ValidateAuth{Valid API Key?}
    ValidateAuth -->|No| AuthError[Return 401 Unauthorized]
    ValidateAuth -->|Yes| Parse

    Parse --> CheckType{Upload Type?}

    CheckType -->|File Upload| ValidateFile[Validate File]
    CheckType -->|URL Ingestion| ValidateURL[Validate URL]

    ValidateFile --> CheckSize{Size < 50MB?}
    CheckSize -->|No| SizeError[Return Size Error]
    CheckSize -->|Yes| CheckMIME{Valid MIME Type?}
    CheckMIME -->|No| TypeError[Return Type Error]
    CheckMIME -->|Yes| PrepareUpload

    ValidateURL --> CheckURLFormat{Valid URL?}
    CheckURLFormat -->|No| URLError[Return URL Error]
    CheckURLFormat -->|Yes| PrepareUpload

    PrepareUpload[Prepare Ingestion Request] --> CallIngest[Call Geneline-X Ingest
API]

    CallIngest --> CreateJob[Create Ingestion Job]
    CreateJob --> JobID[Return Job ID to User]

    JobID --> Poll{Wait for Completion?}
    Poll -->|Yes| PollStatus[Poll Job Status]
    Poll -->|No| End1([Job Started])

    PollStatus --> CheckStatus{Job Status?}
    CheckStatus -->|Pending/Processing| Wait[Wait 5 seconds]
    Wait --> PollStatus

    CheckStatus -->|Completed| Success[Document Ingested Successfully]
    CheckStatus -->|Failed| Failed[Ingestion Failed]

    Success --> End2([Knowledge Base Updated])
    Failed --> ShowError[Show Error Details]
    ShowError --> End3([Process Complete])

    AuthError --> End3
    SizeError --> End3
    TypeError --> End3
    URLError --> End3

    style Start fill:#4CAF50,stroke:#2E7D32,color:#fff
    style CallIngest fill:#2196F3,stroke:#1565C0,color:#fff
```

```
    style Success fill:#4CAF50,stroke:#2E7D32,color:#fff
    style Failed fill:#F44336,stroke:#C62828,color:#fff
```

Detailed Step-by-Step Ingestion Process

Phase 1: Upload Initiation

Step 1: Upload Method Selection You have two ways to start:

- **CLI Tool**: Type a command in your terminal: `npm run ingest file ./document.pdf`
- **Admin API**: Send HTTP POST request to `/admin/ingest/file`

Step 2: Authentication (API Only) If using the API, the system first checks:

- Is the `X-API-Key` header present?
- Does it match the `ADMIN_API_KEY` environment variable?
- If NO: Return 401 Unauthorized, stop here
- If YES: Proceed

Phase 2: Validation

Step 3: Parse Arguments/Request Extract the information:

- File path or URL
- Optional: title, description, category metadata
- Which type of ingestion? (file upload vs URL ingestion)

Step 4: Type-Specific Validation

For File Uploads:

1. **File Size Check**: Is it under 50MB?
    - Why 50MB? Geneline-X API has upload size limits
    - Larger files should use URL ingestion instead
    - If too large: Return error "File size exceeds 50MB, use URL ingestion"
2. **MIME Type Check**: Is it a PDF?
    - Currently only `application/pdf` is supported
    - Future: Could support Word docs, text files, etc.
    - If wrong type: Return error "Only PDF files are supported"
3. **File Exists Check** (CLI only): Does the file path exist?
    - Prevents typos in file paths
    - If not found: Return error "File not found"

For URL Ingestion:

1. **URL Format Check**: Is this a valid URL?
    - Must start with `http://` or `https://`
    - Must be properly formatted
    - If invalid: Return error "Invalid URL format"

Phase 3: Sending to Geneline-X

Step 5: Prepare Ingestion Request Build the request payload:

```
{
  "chatbotId": "your-chatbot-id",
  "file": "<base64 encoded PDF>" OR "url": "https://...",
  "title": "Optional document title",
  "description": "Optional description",
  "category": "Optional category"
}
```

Step 6: Call Geneline-X Ingest API Send POST request to Geneline-X:

- Endpoint: `POST /ingest` (or similar)
- Include authentication
- Upload the file or provide the URL
- Geneline-X receives and acknowledges

Step 7: Create Ingestion Job Geneline-X creates a background job:

- Assigns a unique Job ID (e.g., "job-abc-123-xyz")
- Job status: "pending"
- Returns immediately (doesn't wait for processing)

Why create a job? Processing a 200-page PDF can take minutes. Jobs allow:

- Asynchronous processing
- Status checking
- User doesn't have to wait

Step 8: Return Job ID to User System responds:

```
{
  "success": true,
  "jobId": "job-abc-123-xyz",
  "message": "Ingestion started",
  "status": "pending"
}
```

User now has a Job ID they can use to check status.

Phase 4: Waiting for Completion (Optional)

Step 9: Wait for Completion Decision CLI tool asks: "Should I wait for completion?"

- If `--no-wait` flag: Skip to "Job Started" end, user can check later
- If no flag: Enter polling loop

Step 10: Poll Job Status Every 5 seconds (configurable), ask Geneline-X:

- GET `/ingest/status/job-abc-123-xyz`
- Returns current status:

```
{
  "jobId": "job-abc-123-xyz",
  "status": "processing",
  "progress": 45,
```

```
  "message": "Extracting text from PDF..."
}
```

Step 11: Check Job Status What's the current status?

1. **Pending or Processing**:
    - Status: "pending" or "processing"
    - Action: Wait 5 seconds, poll again
    - Display: "Processing... 45%"
2. **Completed**:
    - Status: "completed"
    - Progress: 100%
    - Message: "Document successfully ingested"
    - Action: Success! Move to completion
3. **Failed**:
    - Status: "failed"
    - Error details: "Could not extract text" or "File corrupted"
    - Action: Show error, end process

Phase 5: Completion

For Successful Ingestion:

1. Document has been processed by Geneline-X
2. Text extracted and indexed
3. Now searchable via `knowledge_search` tool
4. Bot can reference this document in responses
5. Clear success message shown to user

For Failed Ingestion:

1. Show detailed error message
2. Possible causes:
    - Corrupted PDF
    - PDF is scanned image (no text to extract)
    - Network error during processing
    - Geneline-X service issue
3. Suggest solutions:
    - Try re-uploading
    - Convert scanned PDFs to text-based PDFs
    - Check file integrity

What Happens Inside Geneline-X During Processing:

1. **Text Extraction**:
    - Reads PDF file
    - Extracts all text content
    - Preserves structure (headings, paragraphs, lists)
2. **Chunking**:

- Breaks document into logical sections
- Each chunk is 500-1000 words (optimal for search)
- Maintains context within chunks

3. **Embedding**:
   - Converts text chunks into numerical vectors
   - These vectors capture semantic meaning
   - Allows similarity-based search
4. **Indexing**:
   - Stores chunks in vector database
   - Associates with chatbot ID
   - Makes searchable via API
5. **Metadata Storage**:
   - Saves title, description, category
   - Links to source document
   - Enables citation in responses

Real-World Example:

You upload "Standard Treatment Guidelines for Sierra Leone 2021":

1. CLI command: `npm run ingest file ./treatment-guidelines.pdf --title "Treatment Guidelines 2021"`
2. System validates: PDF, 5.49MB (under 50MB), exists
3. Sends to Geneline-X
4. Job created: "job-treatment-001"
5. Processing takes 2 minutes:
   - Extracts 856 pages of text
   - Creates 1,200 searchable chunks
   - Indexes all chunks
6. Status: "completed"
7. Now when user asks "What's the malaria treatment protocol?"
8. Agent calls `knowledge_search` tool
9. Searches through Treatment Guidelines
10. Finds relevant section
11. Cites it in response: "According to the Standard Treatment Guidelines for Sierra Leone 2021, Section 5.2..."

Ingestion Methods Comparison

Method	Use Case	Max Size	Wait for Completion
CLI File Upload	Small files (<5MB), local documents	50MB	Optional (--no-wait flag)
CLI URL Ingestion	Large files, hosted documents	Unlimited	Optional (--no-wait flag)
API File Upload	Programmatic small file upload	50MB	No (async)
API URL Ingestion	Programmatic large file upload	Unlimited	No (async)

Job Status Tracking

```
stateDiagram-v2
    [*] --> Pending: Job Created
    Pending --> Processing: Ingestion Started
    Processing --> Completed: Success
    Processing --> Failed: Error Occurred
    Completed --> [*]
    Failed --> [*]

    note right of Processing
        Status can be checked
        via CLI or API
    end note
```

Component Details

1. WhatsApp Client

Technology: whatsapp-web.js with Puppeteer

```
graph TB
    Init[Initialize Client] --> Puppeteer[Launch Headless Chrome]
    Puppeteer --> Check{Session Exists?}

    Check -->|Yes| Restore[Restore Session]
    Check -->|No| QR[Generate QR Code]

    QR --> Display[Display QR via API]
    Display --> Scan[User Scans with Phone]
    Scan --> Auth[Authenticate]

    Restore --> Ready[Client Ready]
    Auth --> Save[Save Session to .wwebjs_auth]
    Save --> Ready

    Ready --> Listen[Listen for Messages]
    Listen --> Event[Emit Message Events]

    style Init fill:#25D366,stroke:#128C7E,color:#fff
    style Ready fill:#4CAF50,stroke:#2E7D32,color:#fff
```

Features:

- Persistent authentication using LocalAuth
- QR code accessible via `/qr` endpoint
- Automatic session restoration on restart
- Media attachment support (images, documents, audio, video)

2. Rate Limiting & Queue Management

```
graph LR
    subgraph "Per-Chat Queues"
        Q1[Chat A Queue]
        Q2[Chat B Queue]
        Q3[Chat C Queue]
    end

    subgraph "Rate Limiter"
        RL[Rate Limit Tracker]
        T1[Chat A: Last processed at T1]
        T2[Chat B: Last processed at T2]
        T3[Chat C: Last processed at T3]
    end

    subgraph "Worker Pool"
        W1[Worker 1]
        W2[Worker 2]
        W3[Worker 3]
        W4[Worker 4]
        W5[Worker 5]
    end

    Q1 -.->|Check| RL
    Q2 -.->|Check| RL
    Q3 -.->|Check| RL

    RL -->|Time OK| W1
    RL -->|Time OK| W2
    RL -->|Time OK| W3

    style Q1 fill:#4CAF50,stroke:#2E7D32,color:#fff
    style RL fill:#FF9800,stroke:#E65100,color:#fff
    style W1 fill:#2196F3,stroke:#1565C0,color:#fff
```

Configuration:

- Maximum concurrent workers: 5 (configurable via `MAX_CONCURRENCY`)
- Minimum time between messages per chat: 1000ms (configurable via `PER_CHAT_RATE_LIMIT_MS`)
- FIFO ordering within each chat
- Exponential backoff retry for failures

3. Geneline-X AI Integration

```
sequenceDiagram
    participant Bridge as Bridge Service
    participant Client as Geneline Client
    participant API as Geneline-X API

    Bridge->>Client: sendMessage(request)
    Client->>API: POST /message
```

```
    alt Success Response
        API-->>Client: 200 OK + Stream
        Client-->>Bridge: Response Text
    else Rate Limited
        API-->>Client: 429 Too Many Requests
        Client->>Client: Wait (Exponential Backoff)
        Client->>API: POST /message (Retry)
        API-->>Client: 200 OK + Stream
        Client-->>Bridge: Response Text
    else Server Error
        API-->>Client: 5xx Server Error
        Client->>Client: Wait (Exponential Backoff)
        Client->>API: POST /message (Retry)
        API-->>Client: Response
        Client-->>Bridge: Response Text
    else Authentication Error
        API-->>Client: 401 Unauthorized
        Client-->>Bridge: Throw Error
    end
```

Request Structure:

- Chatbot ID
- User email (generated from chat ID)
- Message content (includes system prompt)
- Streaming enabled

4. Topic Restriction System

```
graph TB
    UserMsg[User Message] --> Combine[Combine with System Prompt]

    SysPrompt[System Prompt:<br/>Only answer public health<br/>questions about
Sierra Leone] --> Combine

    Combine --> SendAI[Send to AI]
    SendAI --> AIResponse{AI Evaluates Topic}

    AIResponse -->|On-Topic| Answer[Provide Detailed Answer]
    AIResponse -->|Off-Topic| Decline[Politely Decline:<br/>Ask for health
questions]

    Answer --> User
    Decline --> User

    style SysPrompt fill:#FF9800,stroke:#E65100,color:#fff
    style Answer fill:#4CAF50,stroke:#2E7D32,color:#fff
    style Decline fill:#F44336,stroke:#C62828,color:#fff
```

Configurable via Environment Variables:

- `GENELINE_SYSTEM_PROMPT` : Custom instructions for AI behavior

- Default: Restrict to public health topics in Sierra Leone
- Can be customized for different domains or removed entirely

---

Deployment Architecture

Railway Deployment Flow

```
flowchart LR
    Dev[Developer] -->|git push| GitHub[GitHub Repository]
    GitHub -->|Webhook| Railway[Railway Platform]

    Railway --> Build[Build Docker Image]
    Build --> Dockerfile[Use Dockerfile]
    Dockerfile --> Install[Install Dependencies]
    Install --> Chromium[Install Chromium for Puppeteer]
    Chromium --> TypeScript[Compile TypeScript]
    TypeScript --> Image[Create Container Image]

    Image --> Deploy[Deploy Container]
    Deploy --> Volume[Mount Persistent Volume]
    Volume --> Auth[.wwebjs_auth directory]

    Auth --> Start[Start Application]
    Start --> API[Expose API Endpoints]
    Start --> WA[Connect to WhatsApp]

    WA --> QREndpoint[/qr endpoint available]
    QREndpoint --> Scan[User Scans QR]
    Scan --> SaveSession[Save Session to Volume]

    SaveSession --> Ready[Bot Ready]

    style GitHub fill:#333,stroke:#000,color:#fff
    style Railway fill:#0B0D0E,stroke:#fff,color:#fff
    style Image fill:#2196F3,stroke:#1565C0,color:#fff
    style Ready fill:#4CAF50,stroke:#2E7D32,color:#fff
```

Production Environment

```
graph TB
    subgraph "Railway Container"
        App[Node.js Application]
        Chrome[Headless Chromium]
        Volume[Persistent Volume]
    end

    subgraph "External Services"
        WA[WhatsApp Servers]
        Geneline[Geneline-X API]
```

```
    Search[Web Search API]
end

subgraph "Monitoring"
    Logs[Winston Logs]
    Health[Health Endpoint]
    Status[Status Endpoint]
end

App --> Chrome
App --> Volume
App <-->|Messages| WA
App <-->|AI Requests| Geneline
App <-->|Search Queries| Search

App --> Logs
App --> Health
App --> Status

style App fill:#4CAF50,stroke:#2E7D32,color:#fff
style Volume fill:#FF9800,stroke:#E65100,color:#fff
style Geneline fill:#2196F3,stroke:#1565C0,color:#fff
```

Environment Variables:

- `GENELINE_HOST` : Geneline-X API endpoint
- `GENELINE_API_KEY` : Authentication key
- `GENELINE_CHATBOT_ID` : Specific chatbot identifier
- `ADMIN_API_KEY` : Admin API authentication
- `NODE_ENV` : production
- `MAX_CONCURRENCY` : Worker pool size
- `PER_CHAT_RATE_LIMIT_MS` : Rate limiting

System Features Summary

✅ Implemented Features

Feature	Description	Status
WhatsApp Integration	Full message send/receive with media	✅ Complete
AI Conversation	Geneline-X integration with streaming	✅ Complete
Agent Framework	Multi-step reasoning with tool calling	✅ Complete
Knowledge Search	Query Geneline-X knowledge base	✅ Complete
Web Search	Real-time internet access	✅ Complete
Conversation Memory	Maintains context (10 messages)	✅ Complete
Document Ingestion	PDF upload via CLI and API	✅ Complete

Topic Restriction	Configurable system prompts	✅ Complete
Queue System	Rate limiting and concurrency control	✅ Complete
Admin API	Management and monitoring endpoints	✅ Complete
Logging	Structured Winston logging	✅ Complete
Docker Support	Railway-optimized deployment	✅ Complete

🚀 System Metrics

- **Average Response Time**: 2-5 seconds (depending on AI complexity)
- **Concurrent Users**: Up to 100+ simultaneous conversations
- **Message Throughput**: Limited by rate limiting (1 msg/second per chat)
- **Uptime Target**: 99.9% (with persistent session storage)

Usage Examples

Example 1: Simple Health Question

```
User: What are the symptoms of malaria?

[System Flow]
1. WhatsApp receives message
2. Queue processes message
3. Agent builds prompt with system instructions
4. Geneline-X AI generates response
5. Response sent back to WhatsApp

Bot: Malaria symptoms typically include:
     - High fever
     - Chills and sweating
     - Headache
     - Nausea and vomiting
     ...
```

Example 2: Knowledge Search

```
User: What does the treatment guideline say about managing severe malaria in children?

[System Flow]
1. WhatsApp receives message
2. Agent recognizes need for specific document knowledge
3. Agent calls knowledge_search tool
4. Tool queries Geneline-X knowledge base
5. Results returned to agent
6. Agent formulates response with retrieved information
7. Response sent to WhatsApp
```

```
Bot: Based on the Standard Treatment Guidelines for Sierra Leone 2021:
     For severe malaria in children, the recommended treatment is...
```

Example 3: Web Search for Current Information

```
User: What is the current malaria situation in Sierra Leone?

[System Flow]
1. WhatsApp receives message
2. Agent recognizes need for current information
3. Agent calls web_search tool
4. Tool performs internet search
5. Results returned to agent
6. Agent synthesizes information
7. Response sent to WhatsApp

Bot: According to recent reports from [sources], the current malaria situation...
```

Example 4: Off-Topic Question (Declined)

```
User: What's the weather today?

[System Flow]
1. WhatsApp receives message
2. Agent evaluates against system prompt
3. AI recognizes off-topic request
4. Polite decline response generated

Bot: I'm specifically designed to answer questions about public health topics
     in Sierra Leone. I can help with questions about diseases, vaccinations,
     healthcare facilities, and more. How can I assist you with a health-related
question?
```

API Endpoints Reference

Public Endpoints

Endpoint	Method	Description	Authentication
`/health`	GET	Health check	None
`/status`	GET	Bot status and queue metrics	None
`/qr`	GET	QR code for WhatsApp pairing	None

Admin Endpoints

Endpoint	Method	Description	Authentication
`/send`	POST	Send arbitrary message	Required

`/session/clear`	POST	Clear WhatsApp session	Required
`/queue/stats`	GET	Detailed queue statistics	Required
`/queue/clear`	POST	Clear all queued messages	Required
`/admin/ingest/file`	POST	Upload PDF file	Required
`/admin/ingest/url`	POST	Ingest from URL	Required
`/admin/ingest/status/:jobId`	GET	Check job status	Required
`/admin/ingest/jobs`	GET	List all jobs	Required

Conclusion

System Strengths

✅ **Reliable**: Persistent sessions, retry logic, error handling
✅ **Intelligent**: Multi-step reasoning with tool usage
✅ **Scalable**: Queue system with concurrency control
✅ **Flexible**: Configurable prompts and behavior
✅ **Production-Ready**: Docker support, logging, monitoring

Architecture Highlights

🏗️ **Modular Design**: Clear separation of concerns
🔄 **Event-Driven**: Asynchronous message processing
🛠️ **Extensible**: Easy to add new tools and capabilities
📊 **Observable**: Comprehensive logging and monitoring

End of Presentation