**Agentic AI Chatbot Documentation** (Updated 10th September 2025)

**Project Description**

This project implements an **Agentic AI Chatbot** designed to interact with users, understand queries, perform optional web searches, and crucially, **maintain context across conversations**. The chatbot now supports multiple AI model providers (OpenAI and Groq) through the **LangChain** framework, which allows for agentic orchestration, multi-step reasoning, and tool usage.

**Key Enhancements:**

* **Multi-Provider Support:** The system is no longer limited to OpenAI. It now integrates with **Groq's** high-performance models (like Llama 3), with a flexible architecture to easily add more providers in the future.
* **Persistent Context & Memory:** The chatbot can now remember previous interactions, track variables defined by the user, and maintain a summary of the conversation's context, enabling true multi-turn dialogues.
* **Enhanced Backend (FastAPI):** Includes new endpoints for managing conversations and querying available providers.
* **Advanced Frontend (Streamlit):** Features a redesigned UI with conversation history, session management, and provider-specific configuration.

This setup ensures the chatbot can handle intelligent, context-aware conversations with optional integration of external tools, powered by a choice of leading AI models.

**1. Technical Architecture & Code Explanation**

The system is built on three core files, each with significant upgrades.

**1.1**ai\_agent.py**(The Orchestrator)**

This file defines the AI agent, initializes the LLMs, manages conversation memory, and handles the reasoning process.

**New Imports for Enhanced Features:**

from langchain\_groq import ChatGroq *# NEW: Support for Groq models*

from typing import Optional, Dict, Any, List, Tuple

import uuid

from datetime import datetime

import re *# NEW: For regex pattern matching to extract variables*

**Core New Component:**ConversationManager**Class**  
This class is the heart of the new **Persistent Context** feature. It handles everything related to conversation history and state.

* **Storage:** Saves all conversations to a conversations.json file, making them persistent across server restarts.
* **Structure:** Each conversation is stored with a unique ID, timestamps, full message history, extracted variables, and a context summary.
* **Variable Tracking (**\_extract\_variables**method):** Uses regex pattern matching (e.g., r'\b([A-Za-z\_][A-Za-z0-9\_]\*)\s\*=\s\*([0-9]+(?:\.[0-9]+)?)\b') to automatically identify and store variable assignments (like A=1, price=29.99) from user messages. These variables are then injected into the context of subsequent queries.
* **Context Methods:** Provides methods (get\_full\_conversation\_text, get\_variables\_context) to format the conversation history and variables into a string that can be passed to the LLM, giving it memory and context.

**Multi-Provider Model Initialization**  
The code now dynamically supports multiple providers through a MODEL\_PROVIDERS configuration map.

MODEL\_PROVIDERS = {

"OpenAI": {

"class": initialize\_openai\_model, *# Function to init OpenAI client*

"models": ["gpt-4", "gpt-4o-mini", "gpt-3.5-turbo", "gpt-4o"]

},

"Groq": { *# NEW: Groq provider entry*

"class": initialize\_groq\_model, *# Function to init Groq client*

"models": ["llama3-70b-8192", "llama3-8b-8192", "llama-3.3-70b-versatile"]

}

}

The get\_response\_from\_ai\_agent function uses this map to select the correct initialization function and check for model support, making it easy to add new providers like Anthropic or Mistral in the future.

**Enhanced Agent Workflow with Memory:**

* **Fetches Context:** Before generating a response, the agent retrieves the recent conversation history, any tracked variables, and a saved context summary for the given conversation\_id.
* **Constructs Intelligent Prompt:** It combines the system prompt, the retrieved context, and the user's new query into a single, rich input for the LLM. This is the key to maintaining persistence.
* **Calls the Agent/LLM:** It uses the LangChain agent if search is enabled, or makes a direct, more reliable call to the chosen LLM with the full conversation history to preserve context.
* **Updates Context:** The agent's response is saved to the conversation history. Periodically, it updates the context summary to keep a distilled record of the conversation's purpose.

**New Helper Functions:**

* get\_available\_providers(): Returns a list of supported providers and their models to the frontend.
* clear\_conversation\_history(conversation\_id): Clears the history and variables for a specific conversation.
* get\_conversation\_variables(conversation\_id): Retrieves all variables extracted from a conversation.

**1.2**backend.py**(The API Layer)**

The FastAPI backend has been updated with new endpoints to support the enhanced features.

**New Endpoints:**

* GET /providers: This endpoint calls get\_available\_providers() from the AI agent and returns the list of supported providers and models. This allows the frontend to dynamically populate its selection menus.
* POST /clear\_conversation: This endpoint accepts a conversation\_id and calls the function to clear that conversation's history and state.

**Enhanced**POST /chat**Endpoint:**  
The endpoint now accepts two new fields in the request JSON body:

* conversation\_id: (Optional) The ID of an existing conversation to continue. If not provided, a new one is created.
* provider\_config: (Optional) A dictionary for provider-specific settings, like passing an API key directly from the frontend.

**1.3**frontend.py**(The User Interface)**

The Streamlit UI has been completely overhauled to leverage the new backend capabilities.

**New Features:**

* **Dynamic Provider/Model Selection:** The UI first calls the /providers endpoint to get a list of available options, making the frontend automatically adapt to changes in the backend (e.g., adding a new provider). If the backend is offline, it uses a fallback list.
* **Conversation Management:** A sidebar section shows the active conversation\_id, message count, and a button to **clear the current conversation**, which calls the /clear\_conversation endpoint.
* **Conversation History View:** A checkbox toggles the display of the entire conversation history in the main chat area, providing a clear transcript of the interaction.
* **Provider-Specific Config:** Input fields for entering API keys for OpenAI and Groq directly, which are passed to the backend via the provider\_config parameter.
* **Robust Error Handling:** Improved error handling and user feedback for backend connection issues, timeouts, and other failures.

**2. How to Run the Enhanced Application**

The setup process remains simple but now requires additional API keys.

**1. Environment Setup & Dependencies**

*# Create and activate a virtual environment (recommended)*

python -m venv .myenv

*# On Windows:*

.myenv\Scripts\activate

*# Install packages from requirements.txt*

*# Ensure your requirements.txt includes all necessary packages:*

*# streamlit, fastapi, uvicorn, requests, langchain, langchain-openai, python-dotenv, langchain-groq*

pip install -r requirements.txt

**2. Environment Variables (**.env**file)**  
Create a .env file in your project root with the following variables. The new system requires more keys for multi-provider support.

env

# OpenAI

OPENAI\_API\_KEY=your\_openai\_api\_key\_here

# NEW: Groq

GROQ\_API\_KEY=your\_groq\_api\_key\_here

# For web search tool

TAVILY\_API\_KEY=your\_tavily\_api\_key\_here

**3. Running the Application**  
You must run both the backend and frontend servers.

* **Terminal 1 - Start the Backend (FastAPI):**

uvicorn backend:app --reload --host 0.0.0.0 --port 9999

* **Terminal 2 - Start the Frontend (Streamlit):**

streamlit run frontend.py

The application will open in your browser at http://localhost:8501.

**3. Summary of Enhanced Features**

| **Feature** | **Implementation** | **Benefit** |
| --- | --- | --- |
| **Multi-Provider Support** | A configurable provider map in ai\_agent.py and new initialization functions. | Users can choose between OpenAI and Groq models for different needs (cost, speed, performance). The architecture is modular for easy future expansion. |
| **Persistent Context & Memory** | The ConversationManager class that stores messages, extracts variables, and manages summaries in a JSON file. | The chatbot remembers previous questions, answers, and user-defined variables, enabling complex, multi-step tasks and coherent long-term dialogues. |
| **Variable Tracking** | Regex pattern matching in \_extract\_variables to find assignments like A=1 in user messages. | Allows the user to define context that the AI can reference and compute with later, e.g., "You said the price was $X, what is 15% of that?" |
| **New API Endpoints** | /providers and /clear\_conversation in backend.py. | Enables dynamic frontend configuration and gives users control over their conversation data. |

**4. Limitations & Future Enhancements**

**Current Limitations:**

1. **Basic Search Integration:** The search tool remains a simple stub. A real integration with Tavily or another search API is needed for live web data.
2. **Scalability:** The conversation storage is file-based (conversations.json), which may not be efficient for a large number of simultaneous users. A database (e.g., SQLite, PostgreSQL) would be better.
3. **Context Window Management:** The entire history is sent to the LLM on each turn. For very long conversations, this will exceed the model's context window, necessitating a smart summarization or truncation strategy.

**Future Enhancements:**

1. **Advanced Search:** Integrate the real Tavily Search API or Google Programmable Search.
2. **Database Storage:** Migrate conversation storage from a JSON file to a proper database.
3. **Advanced Memory Strategies:** Implement a more sophisticated memory class that summarizes old interactions dynamically to manage long context windows.
4. **More Providers:** Add support for Anthropic Claude, Mistral, Cohere, and local Ollama models.
5. **Audio & File Upload:** Allow the agent to process uploaded files (PDFs, DOCs, images) and audio inputs.

The chatbot developed here uses **Agentic AI principles**, but it's crucial to understand that it implements a **specific and foundational type of agency**. It's not a fully autonomous agent with complex planning and self-direction; instead, it's a **reasoning and tool-using agent** built on the ReAct paradigm.

**1. Core Principle of Agentic AI: Beyond Simple Chat**

A simple chatbot (like a basic ChatGPT interaction) is **reactive**. You give it a prompt, it generates a response based on its internal knowledge. It doesn't "think" or "use tools."

An **Agentic AI** system, on the other hand, is **proactive and instrumental**. It doesn't just generate a response; it formulates a plan, can use tools (like a calculator, search API, or database) to gather information it doesn't know, reason about the results, and then synthesize a final answer. It's about leveraging the LLM as a **reasoning engine** to decide *what to do* and *how to do it*.

**2. How This Chatbot Implements Agentic AI**

The provided code uses the **LangChain framework** to create this agentic system. Let's break down the key components:

**a. The "Brain" (The LLM)**

The core reasoning engine is the Large Language Model (e.g., GPT-4o or Llama 3 from Groq). This model's key role in the agent is not just to answer, but to **decide the steps needed to answer**.

**b. The "Tools" (The Capabilities)**

The system defines tools that the AI can use. In the ai\_agent.py file, the primary tool is the TavilySearchResults tool.

if allow\_search:

tavily\_tool = get\_tavily\_search\_tool()

tools = [tavily\_tool]

This tool gives the agent the capability to perform web searches. The agent *decides for itself* if it needs to use this tool to answer the user's question. Other tools could be added here, like a Python REPL for calculations, a database query tool, etc.

**c. The "Orchestrator" (The Agent Loop)**

This is the most important part. The initialize\_agent function creates a sophisticated loop that gives the AI its agency. The specific agent type used is "chat-conversational-react-description".

* **ReAct**: This stands for **Reason** + **Act**. This is the agentic loop:
  + **Reason:** The LLM thinks about the current situation (the user's query + the conversation history + any tool results). It decides, "Do I have enough information to answer, or do I need to use a tool?"
  + **Act:** If it needs a tool, it generates the precise command to use that tool (e.g., tavily\_search\_results\_json: "latest news on SpaceX").
  + **Observe:** The result from the tool (e.g., a JSON list of search results) is fed back to the LLM.
  + **Loop:** The LLM **reasons** again: "Okay, based on this new information, can I now answer the user? Or do I need to refine my search or use another tool?" This loop continues until the agent decides it has a sufficient answer.
* This loop is what makes it agentic. The LLM is not just responding; it's *orchestrating a series of actions* to solve a problem.

**d. The "Memory" (The Conversation History)**

A critical aspect of agency is having context. A simple agent might be stateless. Your enhanced system adds **persistent context** via the ConversationManager class.  
The agent doesn't just see the latest message. It receives a carefully constructed prompt that includes:

* The system prompt ("You are a helpful assistant...").
* Recent conversation history.
* Any variables extracted from the history (e.g., A=1).
* A summary of the context.

This allows the agent to make *informed decisions* based on the long-term context of the interaction, making its behavior more coherent and intelligent. It can remember that the user defined a variable three messages ago and use it in a calculation now.

**3. A Concrete Example of the Agentic Loop in Action**

Let's imagine a user asks: "What was the headline of the top result for 'latest AI breakthroughs' last week, and how many words are in that headline?"

A simple chatbot would likely hallucinate an answer. Your Agentic AI would execute a loop similar to this:

* **Reason (Step 1):** *"The user is asking for a specific headline and a word count. I don't have real-time information. I need to use the search tool to find recent news about AI breakthroughs."*
* **Act (Step 1):** It calls the **Tool**: tavily\_search\_results\_json("latest AI breakthroughs last week")
* **Observe:** The tool returns a list of search results with titles and links.
* **Reason (Step 2):** *"Perfect. The top result's title is 'New Model Achieves State-of-the-Art on Reasoning Benchmark'. Now I need to count the words in this string. I can do this with a calculation. I don't need a tool for this simple task."*
* **Act (Step 2):** It performs the internal calculation: headline = "New Model Achieves State-of-the-Art on Reasoning Benchmark", word\_count = len(headline.split()) = 8.
* **Final Answer:** It synthesizes the information: \*"The top headline was 'New Model Achieves State-of-the-Art on Reasoning Benchmark'. This headline contains 8 words."\*

Summary: Why This is Agentic AI

| **Feature** | **Standard Chatbot** | **This Agentic Chatbot** |
| --- | --- | --- |
| **Core Function** | Text generation | **Task completion** through reasoning and action. |
| **Process** | Single step: input → output. | **Multi-step loop:** Reason → Act → Observe → Repeat. |
| **Information** | Relies solely on its pre-trained knowledge. | **Uses external tools** (web search) to access information it doesn't know. |
| **Memory** | Often stateless or short-term context window. | **Persistent, managed memory** with variable tracking and context summarization. |
| **Key LangChain Components** | ChatOpenAI | ChatOpenAI **+** Tool **+** initialize\_agent **+** ReAct framework **+** ConversationManager |

In conclusion, this chatbot is agentic because it uses the LLM as a central reasoning engine to dynamically plan and execute a sequence of actions (using tools) based on its environment (the user's input and conversation history) to achieve a defined goal. The addition of multi-provider support and persistent memory makes this agency more powerful and flexible.