rAgent Framework

The RAgent Framework is a specialized AI agent development platform that makes it easy to own, interact with, and share intelligent resources. Designed for decentralized systems with multiple AI agents (Swarm) working together, RAgent enables seamless integration in Swarm Agent networks.



# 1.Overview of rAgent



## 1.1 What is rAgent ?

RAgent (Resource-Based Agent) is an intelligent, decentralized agent designed to **manage, allocate, and share computational resources** within a multi-agent ecosystem. It transforms various digital assets—such as social media accounts, computing power, data, and execution environments—into functional agents that can be utilized within Swarm-based system.

## 1.2. Why was RAgent created?

RAgent was developed to:

* **Automate resource allocation** – Efficiently distribute and utilize resources without manual intervention.
* **Facilitate resource coordination** – Enable AI agents and systems to share, lend, or exchange computing resources seamlessly.
* **Optimize computational performance** – Ensure maximum efficiency by preventing resource underutilization or wastage.
* **Handle high-demand computational tasks** – Empower AI agents to tackle complex workloads beyond the capability of a single agent.

## 1.3. How does RAgent operate in a decentralized multi-agent (Swarm) system?

RAgent seamlessly integrates into **Swarm-based AI ecosystems**, enabling:

* **Intelligent resource sharing** – Agents dynamically allocate CPU, memory, bandwidth, and storage based on real-time needs.
* **Collaborative task execution** – RAgent distributes workloads across multiple agents, enhancing parallel processing and efficiency.
* **Dynamic resource management** – Agents can request additional resources from peers or locate available capacity within the Swarm.
* **Optimized load balancing** – Advanced algorithms ensure even resource distribution, preventing bottlenecks and maximizing system performance.

# 2. Key Features of RAgent

* **Pre-Built Base Agent** – A foundational AI agent framework that serves as a **template** for creating different types of agents, providing a standardized yet adaptable structure.
* **Built-In RAgents** – A **set of ready-to-use Resource Agents**, developed on top of the Base Agent, allowing seamless integration into various Swarm Agent systems.
* **Custom RAgent Flexibility** – Users can **modify, extend, or create** their own AI Agents by customizing the Base Agent or adapting existing Built-In RAgents to fit specific needs.
* **Conversation Memory Storage** – Enables each AI Agent to retain and utilize **interaction history**, allowing for more contextual and personalized responses.
* **Flexible Agent Responses** – Supports both **Streaming and Non-Streaming** responses, ensuring adaptive, real-time, and efficient interactions with users.

# 3. Key components

RAgent comes in **two distinct types**, each serving different levels of customization and control over resources:

## 3.1. **Built-In rAgents**

### 3.1.1. What are built-in rAgents?

Built-in rAgents are **pre-configured AI agents** developed by our expert team, each designed to integrate seamlessly with specific resource types. These agents provide **automated processing** when users stake resources into the system, ensuring a **plug-and-play** experience within Swarm Agent networks.

Each rAgent type is optimized for a **specific computational function**, enabling **scalability, automation, and decentralized execution** without additional setup.

### 3.1.2. Types of Available rAgents

1. RX (Social Agents): Connects a Twitter (X) account to launch a decentralized Social Agent. Swarms can utilize social agents to spread their memetics, create agentic movements or launch collaborations.
2. RC ( Compute Agent): Run an rClient to launch a Compute rAgent. This can be used by Swarms for edge computation and low-level computation.
3. RD (Data Agent): Run an rClient to launch a Data rAgent. This can be used by Swarms to store data and use different Data services.
4. RE ( Execution Agent): Run a Linux-based rClient to launch an Execution rAgent. Such rAgents allow Swarms to run nodes, applications, agents and other software autonomously.

## 3.2. **Custom rAgents**

### 3.2.1. What are custom agents?

Custom-rAgents allow users to **extend the rAgent framework** by integrating their own resources, computation models, and execution logic. These user-defined agents provide full flexibility beyond the pre-built rAgents, enabling **specialized use cases** and **custom functionalities** tailored to specific needs.

### 3.2.2. Key Features of Custom-rAgents

* **User-Controlled Development** – Users can create, modify, and manage their own rAgents by extending the core framework.
* **Custom Resource Integration** – Providers can integrate **non-standard computing, data, or execution environments** beyond built-in rAgents.
* **Plugin & Method Expansion** – Developers can build **new interaction models, plugins, or automation mechanisms** to enhance agent functionality.
* **Seamless Configuration** – Custom resource parameters and operational logic can be set up via **.env configuration**, ensuring smooth deployment.
* **Decentralized Contribution Model** – Once reviewed for **compatibility, security, and efficiency**, custom rAgents can be officially merged into the broader system.

# **4. Structure of rAgent**

The **rAgent Framework** follows a hierarchical structure that provides modular and extensible components for building and managing AI-powered resource agents. It is designed to facilitate interaction, resource management, and intelligent decision-making within Swarm networks. By adopting a modular design, rAgent enables seamless integration into multi-agent systems while allowing high levels of customization.

## **4.1 Hierarchical Structure**

## 

## 

The rAgent framework is built upon an inheritance model that ensures modularity and extensibility:

* **BaseAgent**: Defines the fundamental attributes and behaviors of an agent, serving as the core class for all agent types.
* **ProviderAgent** (inherits from BaseAgent): Integrates Large Language Models (LLMs) to enable agents to process language-based logic, making them capable of natural language understanding and reasoning.
* **rAgent** (inherits from ProviderAgent): Expands functionalities by adding resource management, tool execution, and integration with external APIs, allowing interaction with specialized data sources or computational services.

### **Class Inheritance Overview:**

Agent (Base class)

↓

ProviderAgent (Abstract class for LLM providers)

↓

├── OpenAIAgent

├── AnthropicAgent

├── rAgent (Base resource agent)

↓

└── RXAgent (Twitter/X specific agent)

## **4.2 Key Features of rAgent**

### **Modularity and Scalability**

* The framework is designed to be modular, ensuring each agent can be extended or modified without affecting others.
* Scalability allows multiple agents to interact and operate concurrently within a system.

### **Extensibility**

* Developers can create new agents that inherit from rAgent and expand their functionality.
* Supports resource management for efficient interaction with databases, APIs, and other systems.

## **4.3 BaseAgent**

**BaseAgent** provides core functionalities and attributes common to all agents.

### **Key Attributes:**

* name: Unique identifier for the agent.
* description: Brief overview of the agent's purpose.
* save\_chat: Boolean flag to enable/disable memory storage.
* callbacks: Handles interactions during execution, enabling event-driven behavior.
* share\_global\_memory: Determines if the agent shares memory across multiple instances for collaborative processing.
* character: Defines the persona or behavioral style of the agent when responding.
* in\_memory: Enables agents to store temporary data for session-based interactions.

### **Core Method:**

* process\_request(input\_text, user\_id, session\_id, chat\_history, additional\_params)
  + Abstract method that must be implemented by subclasses. It processes user input, maintains contextual history, and generates a response via the appropriate logic or model interaction.

### **BaseAgent Class:**

class BaseAgent(ABC):

def \_\_init\_\_(self, options: AgentOptions):

self.name = options.name

self.description = options.description

self.save\_chat = options.save\_chat

self.callbacks = options.callbacks or AgentCallbacks()

self.share\_global\_memory = options.share\_global\_memory or False

@abstractmethod

async def process\_request(self, input\_text: str, user\_id: str, session\_id: str,

chat\_history: List[ConversationMessage], additional\_params: Dict[str, Any] = None)

-> Union[ConversationMessage, AsyncIterable[Any]]:

pass

## **4.4 ProviderAgent**

**ProviderAgent** extends BaseAgent by integrating LLMs for advanced language processing. This enables the agent to generate responses, perform complex reasoning, and utilize external knowledge sources.

### **Additional Attributes:**

* streaming: Enables streaming responses, allowing the agent to return partial results in real-time.
* inference\_config: Fine-tunes response generation settings such as temperature, max tokens, and top-p filtering.
* extra\_tools: List of additional tools that extend the agent’s functionality beyond LLM interaction.
* llm\_params: Custom parameters for connecting and configuring LLMs, making it adaptable to different models.

### **Principal Methods:**

* initialize\_client()
  + Initializes the LLM client based on provided parameters and authentication credentials.
* handle\_single\_response(request\_options)
  + Manages synchronous responses from external LLMs, ensuring deterministic output.
* handle\_streaming\_response(request\_options)
  + Enables efficient handling of real-time, incremental outputs from streaming LLM responses.
* \_configure\_tools(extra\_tools)
  + Registers and configures tools available to the agent

For specific implementations, refer to:

* [OpenAIAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)
* [AnthropicAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)
* [BedrockAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)
* [LexBotAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)

class ProviderAgent(BaseAgent, ABC):

def \_\_init\_\_(self, options: ProviderAgentOptions):

super().\_\_init\_\_(options)

self.streaming = options.streaming or False

self.inference\_config = options.inference\_config or {}

self.extra\_tools = options.extra\_tools or []

self.llm\_params = options.llm\_params or {}

self.client = self.initialize\_client()

self.\_configure\_tools()

def initialize\_client(self):

if self.llm\_params.get("provider") == "openai":

return OpenAI(\*\*self.llm\_params)

elif self.llm\_params.get("provider") == "anthropic":

return AnthropicClient(\*\*self.llm\_params)

else:

raise ValueError("Unsupported LLM provider")

def is\_streaming\_enabled(self) -> bool:

return self.streaming is True

@abstractmethod

async def handle\_single\_response(self, request\_options: Dict[str, Any]) -> Any:

pass

@abstractmethod

async def handle\_streaming\_response(self, request\_options: Dict[str, Any]) -> Any:

pass

## **4.5 rAgent (Resource Agent)**

**rAgent** is a specialized class designed to manage and interact with external resources, such as databases, APIs, or computational services. This enhances the agent’s ability to operate autonomously while leveraging external data sources.

### **Critical Methods:**

* connect\_resource()
  + Establishes secure, persistent connections to external resources such as databases, APIs, or cloud services.
* validate\_resource\_access()
  + Ensures that authentication credentials are valid and sufficient for accessing the designated resources.
* refresh\_authentication()
  + Periodically refreshes authentication tokens or credentials to maintain uninterrupted access to protected resources.
* \_configure\_tools(extra\_tools)
  + Registers and configures tools necessary for interacting with external resources.

### **Additional Attributes:**

* resource\_id: Unique identifier for the resource.
* resource\_type: Specifies the type of resource (e.g., database, compute node, social media).
* resource\_auth: Credentials or authentication details for accessing protected resources.
* resource\_config: Configuration settings specific to the resource.
* resource\_tools: Tools designed specifically for direct interaction with the resource.
* extra\_tools: Additional external tools that can be used in conjunction with the resource tools.

For specific implementations, refer to:

* [RXAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)
* [RDAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)
* [RCAgent](https://chatgpt.com/c/67c9f080-47c8-8003-a6c8-2cbc20b2e2c8#)

### **rAgent Code:**

class rAgent(ProviderAgent):

def \_\_init\_\_(self, options: rAgentOptions):

super().\_\_init\_\_(options)

self.resource\_id = options.resource\_id

self.resource\_type = options.resource\_type

self.resource\_auth = options.resource\_auth or {}

self.resource\_config = options.resource\_config or {}

self.resource\_tools = AgentTools(tools=[])

self.\_configure\_tools(options.extra\_tools)

async def connect\_resource(self) -> bool:

"""Establishes a secure connection with the external resource."""

return True

def \_configure\_tools(self, extra\_tools: Optional[Union[AgentTools, list[AgentTool]]]):

interact\_func = AgentTool(

name="interact\_with\_resource",

description="Executes predefined operations with the external resource.",

properties={"param": {"type": "string", "description": "Operation-specific parameters."}},

func=self.resource\_interaction\_function

)

self.resource\_tools.tools.append(interact\_func)

if extra\_tools:

self.resource\_tools.tools.extend(extra\_tools)

## **4.6 Creating Custom Agents**

Developers can extend the framework by implementing custom rAgents tailored to their specific needs.

### **Steps to Customize an rAgent:**

1. **Inherit from rAgent** to define a new resource agent.
2. **Specify unique attributes** related to the custom resource.
3. **Implement necessary methods** such as connect\_resource and process\_request.
4. **Register the new agent** within the Swarm ecosystem.

### **Example: Creating a Custom Database Agent**

from typing import Dict, Any, List

from dataclasses import dataclass

from multi\_agent\_orchestrator.agents import rAgent, rAgentOptions

from multi\_agent\_orchestrator.types import ConversationMessage

@dataclass

class DatabaseAgentOptions(rAgentOptions):

database\_url: str

credentials: Dict[str, Any]

class DatabaseAgent(rAgent):

def \_\_init\_\_(self, options: DatabaseAgentOptions):

super().\_\_init\_\_(options)

self.database\_url = options.database\_url

self.credentials = options.credentials

async def connect\_resource(self) -> bool:

"""Establish a connection to the database."""

print(f"Connecting to database at {self.database\_url}")

return True

async def query\_database(self, query: str) -> List[Dict[str, Any]]:

"""Executes a database query and returns results."""

print(f"Executing query: {query}")

return [{"result": "Sample Data"}]

async def process\_request(

self, input\_text: str, user\_id: str, session\_id: str,

chat\_history: List[ConversationMessage], additional\_params: Dict[str, Any] = None

) -> ConversationMessage:

"""Process user requests by executing database queries."""

response = await self.query\_database(input\_text)

return ConversationMessage(role="assistant", content=[{"text": str(response)}])

### **Registering a Custom Agent**

Once the custom agent is implemented, it needs to be registered in the system to be recognized and invoked within the Swarm framework.

# Example registration of DatabaseAgent

db\_agent\_options = DatabaseAgentOptions(

database\_url="https://mydatabase.com",

credentials={"user": "admin", "password": "securepass"}

)

db\_agent = DatabaseAgent(db\_agent\_options)

register\_agent(db\_agent)

By following these examples, developers can quickly extend the rAgent framework to create specialized agents that interact with different resources, making the framework highly adaptable for various AI-driven applications.

## **4.7 Conclusion**

The **rAgent Framework** provides a robust, modular, and extensible structure for AI-driven agents. With built-in support for LLMs, external resource interactions, and tool execution, it serves as a flexible foundation for developers to build customized AI agents suited to various applications. The framework’s hierarchical approach ensures scalability and ease of expansion, making it a powerful solution for intelligent automation in multi-agent environments.

# **5. Conversation Memory Storage**

Effective memory management is crucial for the rAgent Framework, as agents must maintain context-aware interactions across multiple conversations. The framework employs a hybrid memory storage approach, combining global centralized memory with local agent-specific memory. This design ensures optimal scalability, seamless multi-agent collaboration, and efficient memory retrieval.

## **5.1 Hybrid Memory Architecture**

The memory system in rAgent consists of two key components:

### 1. Global Memory (Centralized Storage)

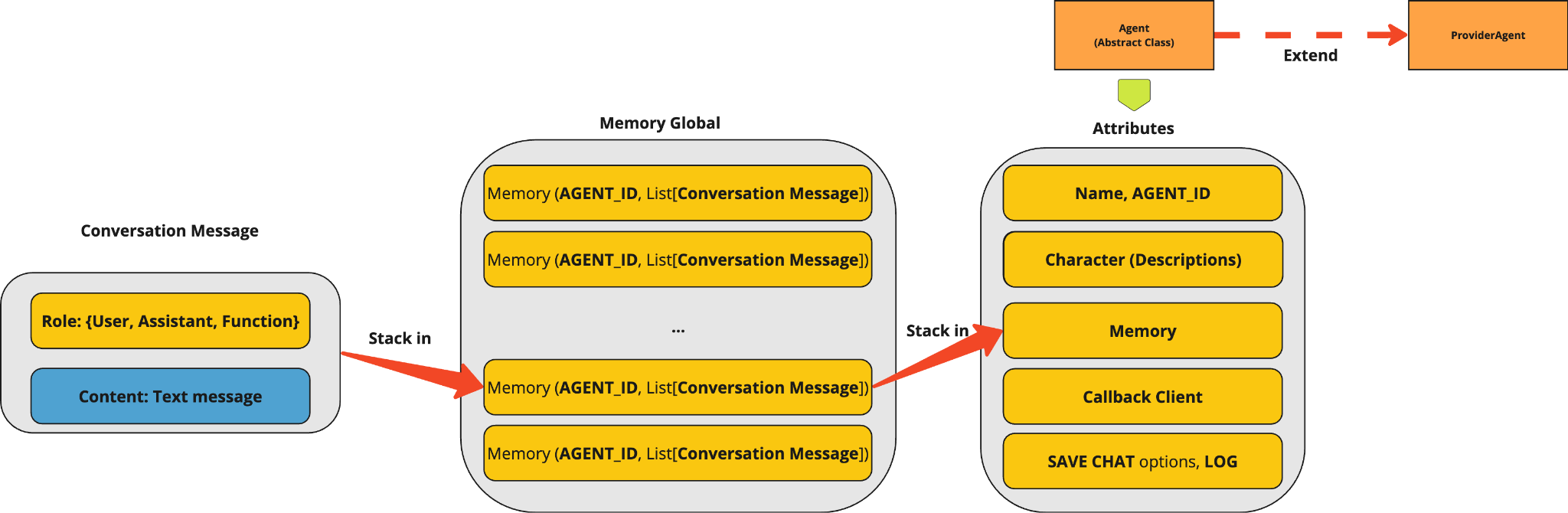
* Stores conversation histories across all agents in a structured manner.
* Maintains consistency across different user sessions and Swarm AI networks.
* Provides a shared knowledge base, enabling long-term contextual awareness.

### 2. Agent-Level Memory (Decentralized & In-Memory Storage)

* Each agent fetches relevant conversation history from Global Memory upon initialization.
* Local agent memory caches recent interactions for fast real-time processing.
* Updates occur both in-memory and globally, ensuring synchronization between agents.

This hybrid approach balances:

* Centralized memory for shared knowledge.
* Local in-memory storage for quick response generation.



### Key Advantages of Hybrid Storage

✔ Efficient Memory Synchronization – Ensures context retention across sessions.  
 ✔ Scalability – Supports multiple agents operating in parallel.  
 ✔ Decentralized Processing – Agents retrieve relevant memory while contributing to a shared knowledge base.  
 ✔ Big Data-Inspired Storage – Uses central indexing + distributed caching, similar to Big Data models.

## 5.2 Memory Structure Overview

The Conversation Memory Storage system is implemented as a hierarchical memory stack, as illustrated in the diagram above.

### Conversation Message Structure

Each conversation consists of a sequence of messages exchanged between:

* User
* Assistant (Agent)
* Function Calls

Messages contain:

* Role (User, Assistant, Function)
* Content (text-based interaction)

Each agent memory stack maintains:

1. Recent Messages for real-time inference.
2. Historical Memory fetched from Global Storage.

This architecture ensures agents retain contextual memory, optimize information retrieval, and enhance collaboration in a multi-agent environment.

For a detailed breakdown of implementation, refer to the Memory Storage Documentation here. 🚀

# **6. Example & Quick Start**

This section provides step-by-step instructions to quickly run rAgent. Whether in a simple console-based interaction or a web-based UI using Chainlit, these examples will help you deploy and test your agent effortlessly.

## **6.1 Setting Up Your Environment**

Before running rAgent, it’s highly recommended to set up a virtual environment to isolate dependencies.

### **Step 1: Create and Activate a Virtual Environment**

Run the following commands in your terminal:

# Create a virtual environment

python -m venv venv

# Activate the virtual environment (MacOS/Linux)

source venv/bin/activate

# Activate the virtual environment (Windows)

venv\Scripts\activate

### **Step 2: Install Dependencies**

Ensure rAgent and required libraries are installed:

```console

pip install ragent chainlit python-dotenv

```

Now you’re ready to run the agent!

## **6.2 Run rAgent in Console**

To quickly test an rAgent, you can run it in console mode, where the agent directly processes user input and responds in the terminal.

### **Example: Running an RXAgent (Twitter/X Agent)**

Create a Python script (run\_rx\_agent.py) with the following code:

**import asyncio**

**import uuid**

**from r\_agent.rx\_agent import RXAgent, RXAgentOptions**

**from multi\_agent\_orchestrator.types import ConversationMessage**

**# Initialize RXAgent with options**

**options = RXAgentOptions(**

**name="X Agent",**

**description="Handles interactions with Twitter/X, including posting tweets.",**

**api\_key="your-api-key",**

**model="gpt-4o",**

**base\_url="https://api.openai.com",**

**xaccesstoken="your-twitter-api-token",**

**inference\_config={**

**"maxTokens": 500,**

**"temperature": 0.5,**

**"topP": 0.8,**

**"stopSequences": []**

**},**

**tool\_config={**

**"tool": "Xtools",**

**"toolMaxRecursions": 5**

**}**

**)**

**agent = RXAgent(options)**

**async def run\_console():**

**user\_id = str(uuid.uuid4())**

**session\_id = str(uuid.uuid4())**

**print("Welcome to RXAgent Console. Type 'exit' to quit.")**

**while True:**

**user\_input = input("\nYou: ").strip()**

**if user\_input.lower() == "exit":**

**print("Goodbye!")**

**break**

**chat\_history = [] # Normally, you'd fetch previous conversation history**

**response = await agent.process\_request(**

**input\_text=user\_input,**

**user\_id=user\_id,**

**session\_id=session\_id,**

**chat\_history=chat\_history**

**)**

**if isinstance(response, ConversationMessage):**

**print(f"\n{response.content[0]['text']}")**

**# Run the agent in async mode**

**if \_\_name\_\_ == "\_\_main\_\_":**

**asyncio.run(run\_console())**

### **Run the Script**

**python run\_rx\_agent.py**

You can now interact with the agent in a text-based console.

## **6.3 Run rAgent with Chainlit**

For a web-based chat experience, rAgent can be integrated with Chainlit, a lightweight chatbot framework.

### **Example: Running RXAgent in Chainlit**

Create a new Python script (run\_chainlit.py) and add the following code:

**import uuid**

**import chainlit as cl**

**import os**

**from dotenv import load\_dotenv**

**# Import RXAgent**

**from r\_agent.rx\_agent import RXAgent, RXAgentOptions**

**from multi\_agent\_orchestrator.types import ConversationMessage**

**# Load environment variables**

**load\_dotenv()**

**# Create RXAgent**

**def create\_X\_agent():**

**options = RXAgentOptions(**

**name="X Agent",**

**description="Handles Twitter/X interactions such as posting tweets.",**

**api\_key=os.getenv("API\_KEY"),**

**model="gpt-4o",**

**base\_url="https://api.openai.com",**

**xaccesstoken=os.getenv("TWITTER\_ACCESS\_TOKEN"),**

**inference\_config={**

**"maxTokens": 500,**

**"temperature": 0.5,**

**"topP": 0.8,**

**"stopSequences": []**

**},**

**tool\_config={**

**"tool": "Xtools",**

**"toolMaxRecursions": 5**

**}**

**)**

**return RXAgent(options)**

**agent = create\_X\_agent()**

**@cl.on\_chat\_start**

**async def start():**

**cl.user\_session.set("user\_id", str(uuid.uuid4()))**

**cl.user\_session.set("session\_id", str(uuid.uuid4()))**

**@cl.on\_message**

**async def handle\_message(message: cl.Message):**

**user\_id = cl.user\_session.get("user\_id")**

**session\_id = cl.user\_session.get("session\_id")**

**msg = cl.Message(content="")**

**await msg.send()**

**cl.user\_session.set("current\_msg", msg)**

**response = await agent.process\_request(**

**input\_text=message.content,**

**user\_id=user\_id,**

**session\_id=session\_id,**

**chat\_history=[]**

**)**

**if isinstance(response, ConversationMessage):**

**await msg.stream\_token(response.content[0]["text"])**

**await msg.update()**

**# Run Chainlit server**

**if \_\_name\_\_ == "\_\_main\_\_":**

**cl.run()**

### **Run the Chainlit Server**

**chainlit run run\_chainlit.py**

A web interface will open, allowing you to chat with RXAgent in real-time.

**7. Contribution Progress**

We value your contributions! Before submitting changes, please start a discussion by opening an issue to share your proposal.

Once your proposal is approved, here are the next steps:

1. 📚 Review our [Contributing Guide](https://github.com/awslabs/multi-agent-orchestrator/blob/main/CONTRIBUTING.md)
2. 💡 Create a [GitHub Issue](https://github.com/awslabs/multi-agent-orchestrator/issues)
3. 🔨 Submit a pull request

✅ Follow existing project structure and include documentation for new features.

🌟 Stay Updated: Star the repository to be notified about new features, improvements, and exciting developments in the Multi-Agent Orchestrator framework!