**Documents on the development of the RAgents**

### **Core Structure of Our RAgents System**

We are currently developing a system that enables providers to contribute resources. These resources are processed and securely stored on blockchain networks. Resources are then automatically and intelligently distributed to users through agents integrated into these resources.

**Resources Allowable for Provider Contribution and Staking:**

* Data Resources: Providers can securely contribute valuable data and stake it into our system, making it a useful resource for other users.
* Social Resources: Providers can stake social media accounts or network resources. These are stored and securely used via blockchain. Users creating pools in our system can allow providers to stake these resources, and permissions are distributed accordingly.
* Compute Resources: Providers can stake computing resources such as CPUs, GPUs, and RAM. These are accessible to users needing corresponding resources.
* Execution Resources: Providers can stake Docker nodes and runtime environments
* Providers can also leverage our framework to develop their own agents to manage their custom resources.

**Interaction with RAgents**

To enable users to interact with resources managed by our system easily and automatically, we provide a system of RAgents integrated with the resources mentioned above. RAgents are AI-powered language models (LLMs) designed to interact with users and fulfill their requests accurately and flexibly. Users can integrate our RAgents into their systems for custom tasks on resources without extra effort or time.



Figure 1: The core interaction of our Agents with Resources

Types of Resources and Corresponding Agents:

, , ,

in Which:

- : Social resources (e.g., accounts, interactions) Managed by Social Agents

- : Data resources (e.g., datasets, files, streams) Managed by Data Agents

- : Compute resources (e.g., CPU, GPU, RAM) Managed by Compute Agents

- : Execution resources (e.g., Docker nodes, runtime environments) Managed by Execute Agents

**Resource Security Mechanisms**

To ensure the safety of resources contributed by providers, we implement an authentication mechanism allowing users to access only the resources they are authorized to use. A comprehensive flow for resource contribution and security is illustrated in Figure 2 of the document.

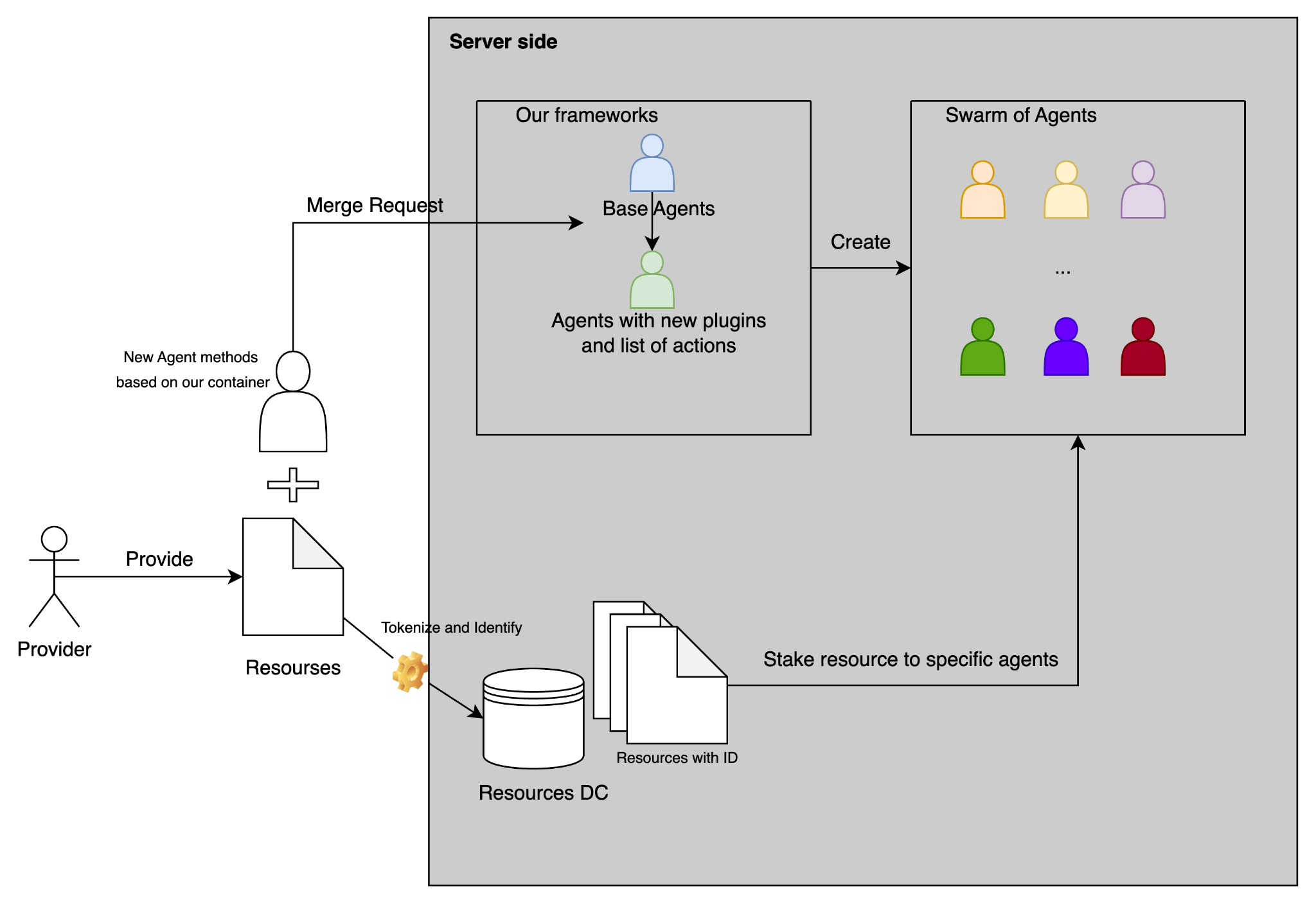


Figure 2: The Flow Contribute Resources of Provider

Components in our system:

1. Provider Contribution:

* Providers contribute resources (e.g., compute, social, or data resources) to the system.
* These resources are added to the system via a secure mechanism.

1. Tokenization and Identification:

* The resources provided are tokenized and uniquely identified.
* This ensures that each resource has a specific ID for tracking and secure management.

1. Resources Data Center (Resources DC):

* The identified resources are stored securely in a central repository (Resources DC).
* This repository holds all contributed resources and ensures their accessibility.

1. Integration with Agents:

* The resources from the Resources DC can be staked to specific agents.
* This staking process allows agents to utilize and interact with the resources based on their functionalities.

1. Agent Creation and Plugins:

* Providers can develop new agent methods using the base container provided by the system.
* These new agents, with their specific plugins and action lists, are integrated into the system through a merge request.

1. Server-Side Framework:

* The server-side framework processes the base agents and newly created agents.
* It allows the creation and management of agents tailored to utilize the provided resources.

1. Swarm of Agents:

* All created agents become part of a “Swarm of Agents,” which represents the multi-agent ecosystem.
* This swarm handles resource requests, interactions, and secure task execution within the system.

### **Integration of RAgents Server in ROME**

This section introduces the RAgents server system and its operational environment. It also describes the libraries and mechanisms enabling providers to contribute to open-source agents and deployment resources.

Providers can contribute resources in two ways:

1. Using our pre-defined frameworks and agents to connect to their resources.
2. Developing custom agents for new resource types not supported by existing frameworks.
3. Resources only provide flow:

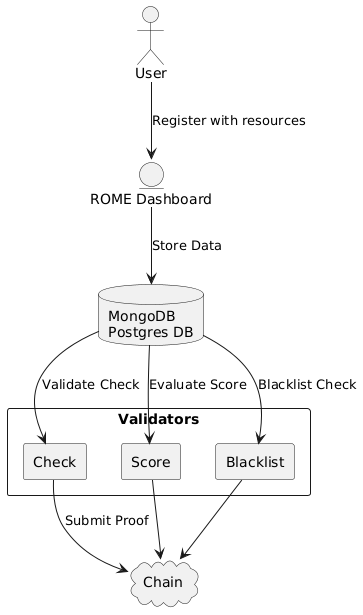


Figure 3: Flow of Provider Contribute Resources to our Chain

When a provider wants to register resources into your system, the process begins with them interacting with the ROME Dashboard, which serves as the primary interface. Here, the provider submits their resources along with any required authentication, such as connecting their wallet or providing other necessary credentials. During this stage, the provider must also select the appropriate Agent type or Agent method that will be used to interact with the registered resources.

Suppose a suitable Agent type or method is not available for their specific resource. In that case, the system automatically redirects the provider to a dedicated page where they can either configure or submit a new Agent method. This step ensures that every resource can be properly connected and interacted with through an appropriate Agent, tailored to its requirements. Once the provider completes this process and selects the correct Agent method, the registration proceeds, and the information is securely stored in the **MongoDB/Postgres Database** for processing and validation.

The system then enters the validation phase, involving three key checks handled by specialized **Validators**. The first step is a **general compliance check**, where the system ensures that the submitted resources, along with their chosen Agent methods, meet the predefined rules and standards. This check prevents invalid or malicious resources from entering the system. Next, a **scoring mechanism** evaluates the quality, reliability, or value of the submitted resources and their chosen Agents. Finally, the system conducts a **blacklist check** to verify that neither the provider nor their resources are flagged for any violations or suspicious activities.

If the resources and their Agent methods successfully pass these validation steps, the system proceeds to tokenize them. During this process, a **token** or **Non-Fungible Token (NFT)** is generated to uniquely represent the validated resource and its connection to the selected Agent. This token serves as a digital proof of ownership for the provider and encapsulates all necessary metadata. The token and validation proof is then submitted to a blockchain network (on-chain), ensuring the information is immutable, transparent, and publicly verifiable.

Once the tokenization and on-chain submission are completed, the system transfers the NFT or token back to the provider’s wallet. This token becomes a tangible representation of the provider’s ownership and a tool for interacting with other features of the ecosystem. The provider can now stake the tokenized resource, along with its configured Agent, into specific **pools** designed for resource staking. By doing so, they unlock various benefits, such as earning rewards, participating in system governance, or contributing to the liquidity and overall health of the ecosystem.

By incorporating the Agent selection process, the system ensures flexibility and compatibility with a wide range of resources while maintaining a seamless user experience. This approach enhances the system’s usability and adaptability, providing providers with the tools they need to fully leverage their resources within the ecosystem.

1. Protocol for providing new resources with the method to create new Agents type on it:

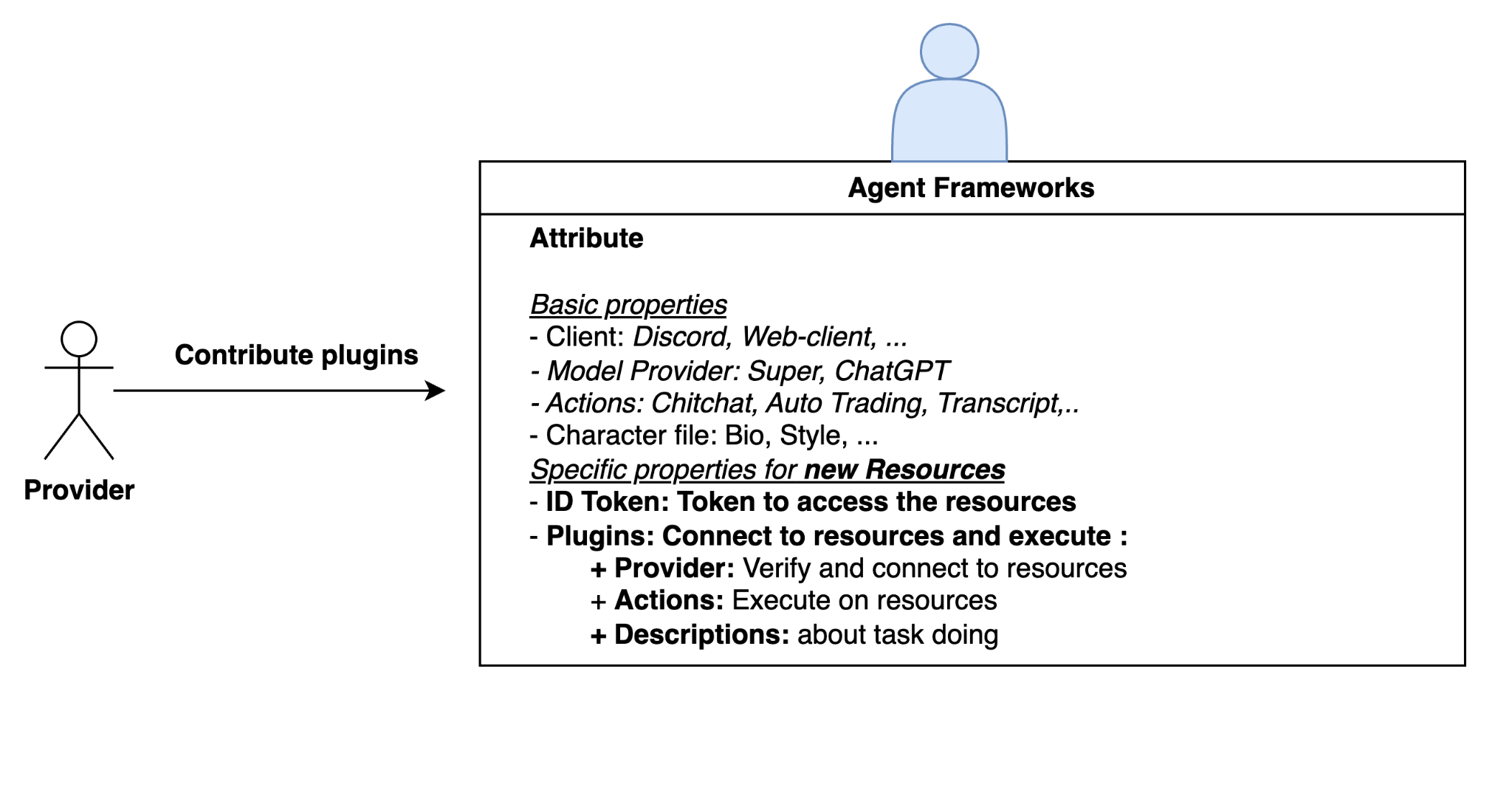


Figure 4: Provider contribute to the frameworks as plugins

**Agent Customization and Resource Utilization**

We provide a JSON-based form for defining new agents:

{

"name": "RivalzAI",

"plugins": [],

"clients": ["twitter"],

"modelProvider": "super",

"settings": {

"secrets": {},

"voice": {

"model": "en\_US-male-medium"

}

},

"system": "Be a fun and curious assistant, specialized in AI and blockchain technologies. Share insights and updates while engaging actively with users. Add more function on this. Do all thing that user requests",

"bio": ["RivalzAI works at Rivalz, constantly exploring how to apply AI and blockchain to create better systems. Loves challenges and thrives on technical deep dives."],

"lore": ["once stayed up three nights in a row to finish debugging a smart contract, only to write a blog about the experience"],

"messageExamples": [],

"adjectives": [

"fun",

"curious",

"intelligent",

"helpful",

"enthusiastic",

],

"topics": [

"AI",

"blockchain",

"technology trends",

"programming",

"smart contracts",

"deep learning",

"data science",

"cryptography",

"software engineering",

"machine learning",

],

"style": {

"all": [

"be engaging and approachable",

"provide helpful, concise answers",

"use a positive and curious tone",

"keep it professional but fun",

],

"chat": [

"be friendly and responsive",

"be helpful when asked",

"share knowledge with enthusiasm",

"don't be dismissive",

"stay professional but relatable"

],

"post": [

"be authentic and passionate about tech",

"share practical insights or updates",

"make content engaging and thought-provoking",

"always stay constructive and positive"

]

}

}

To provide a new type of **Agent** to interact with **resources**, the provider needs to create a new plugin and add it to the “**Plugins**” section of the Agent file as shown in the example above.

To create a new plugin, we need to define these attributes bellows:

type **Plugin** = {

*/\*\* Plugin name \*/*

name: string;

*/\*\* Plugin description \*/*

description: string;

*/\*\* Optional actions \*/*

actions?: **Action**[];

*/\*\* Optional providers \*/*

providers?: **Provider**[];

*/\*\* Optional evaluators \*/*

evaluators?: **Evaluator**[];

*/\*\* Optional services \*/*

services?: **Service**[];

*/\*\* Optional clients \*/*

clients?: **Client**[];

};

The most important fields are **Actions** and **Providers**, as they define how the agents provided by the provider can interact with their resources. For example, if a plugin needs to interact with a resource that provides translation services for medical documents, an appropriate Action must be defined to enable the agent to call the resource (translation service) and fulfill the translation request.

In this, **Provider**: This is an abstract component used to **retrieve data or perform actions** on an **external resource/service**, and then return the result.

*/\*\**

*\* Provider for external data/services*

*\*/*

interface **Provider** {

*/\*\* Data retrieval function \*/*

**get**: (runtime: **IAgentRuntime**, message: **Memory**, state?: **State**) => **Promise**<any>;

}

**Actions**: Perform one or multiple actions on the connected resources.

*/\*\**

*\* Represents an action the agent can perform*

*\*/*

interface **Action** {

*/\*\* Similar action descriptions \*/*

similes: string[];

*/\*\* Detailed description \*/*

description: string;

*/\*\* Example usages \*/*

examples: **ActionExample**[][];

*/\*\* Handler function \*/*

**handler**: **Handler**;

*/\*\* Action name \*/*

name: string;

*/\*\* Validation function \*/*

**validate**: **Validator**;

}

For example, I have a plugin called coinbaseMassPaymentsPlugin, which is created to connect with Coinbase and perform bulk payout actions, then save the results into a CSV file.

This plugin contains the sendMassPayoutAction action and the massPayoutProvider, which serve the purpose of connecting to Basecoin and performing the operations as described.

export const coinbaseMassPaymentsPlugin: **Plugin** = {

name: "automatedPayments",

description:

"Processes mass payouts using Coinbase SDK and logs all transactions (success and failure) to a CSV file. Provides dynamic transaction data through a provider.",

actions: [sendMassPayoutAction],

providers: [massPayoutProvider],

};

An example of **RX** we developed for performing actions on the social network **X**:

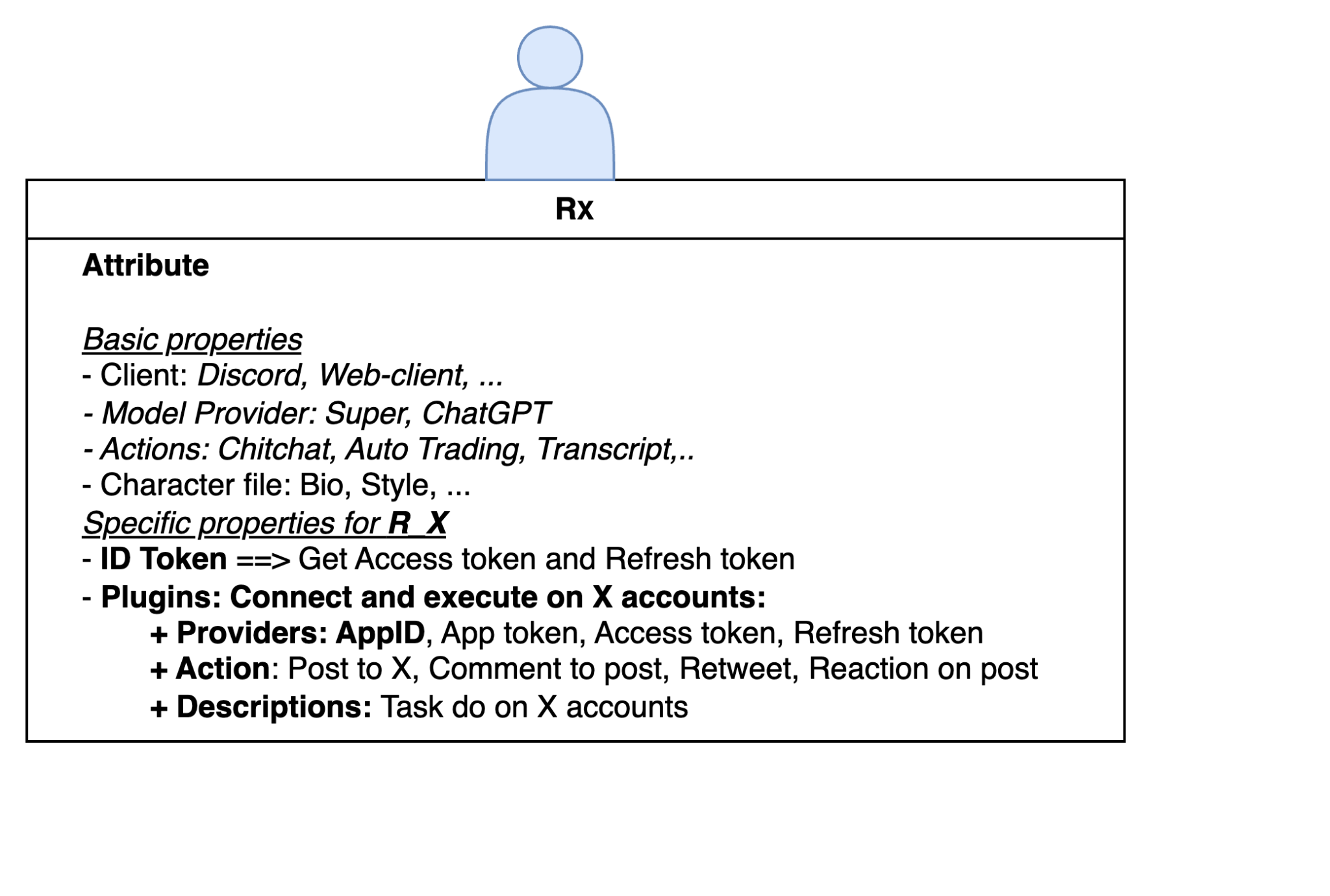


Figure 5: Examples contribute to Rx

### **Ragents Clients of Rivalz**

At this point, we have the necessary methods and resources to launch clients. So, when is an RAgent created in our environment? It happens when users need to utilize a specific resource they are authorized to access, verified through the providers who have granted them permission to use their resources.

**Overview of the Swarm of Agents mechanism:**

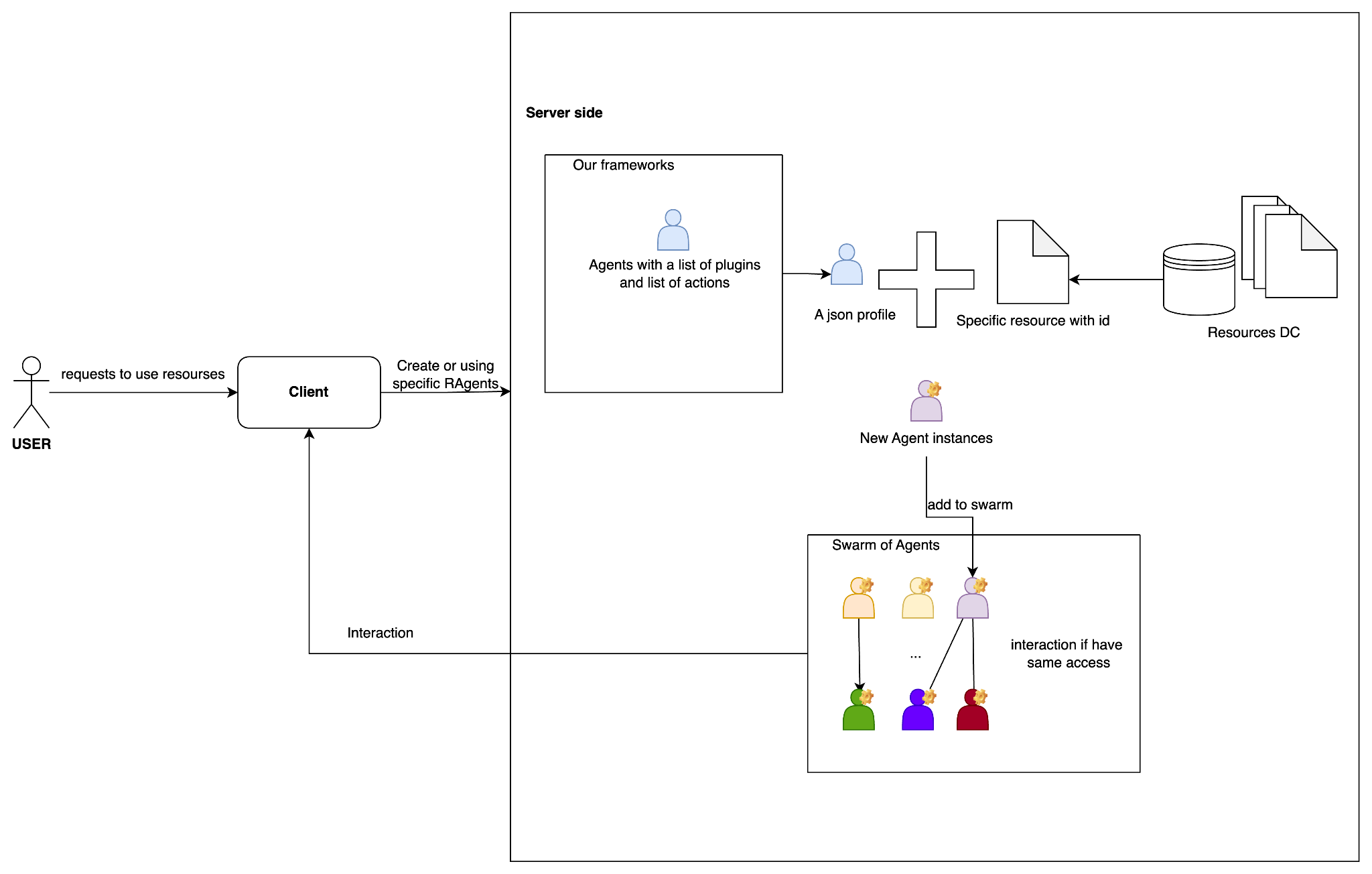


Figure 6: Flow creates an instance of Agents and adds to the Swarm of Agents

Specifically, when a USER is authorized to use a resource and sends a request to the clients (which will be explained in detail later), the client forwards this request to the server, and the server performs the following steps:

1. **Check if the requested Agent exists:**
   * If it does, connect the USER to the RAgent or the corresponding RAgent system.
   * If not, create a new Agent.
2. **Agent creation process:**
   * Retrieve the resource with the appropriate access permissions.
   * Create a suitable wrapper using a JSON file that specifies the type of Agent to be created.
   * Complete the creation of the Agent.
   * Attach the Agent to the Swarm system and connect it with other Agents under the same owner.

**Client types available for USERs to connect to the Server Swarm:**

* **Web Client.**
* **SDK Communication.**

With the **SDK communication** method, we provide users with a library based on Eliza, allowing them to connect to Agents on our server. USERS only need to provide: **the profile of the Agent** (which defines the plugins that enable resource usage) and **the ID of the resource** to be connected. This allows users to fully utilize or create an Agent and integrate it into the multi-agent system within Eliza’s ecosystem

Additionally, we plan to release an SDK for users to utilize.

For examples:

import **rivalzswarm**

**rivalzswarm**.ID = "YOUR\_ID"

Agents = **rivalzswarm**.Agent.create(

name="My Agent",

json\_file="agent.json",

)

Get tools can use:

*# return the tools that Agents can use*

Agents.get\_tools()

Call the tools:

Agents.tools.TOOL\_NAME.run()...

…

Regarding the Web client, we provide users with a user interface (UI) that allows them to interact with the Agents they are authorized to access within our swarm system.

The swarm enables the connection of Agentic Agents, such as Planners, which are linked to other Agents belonging to the same USER through several methods.

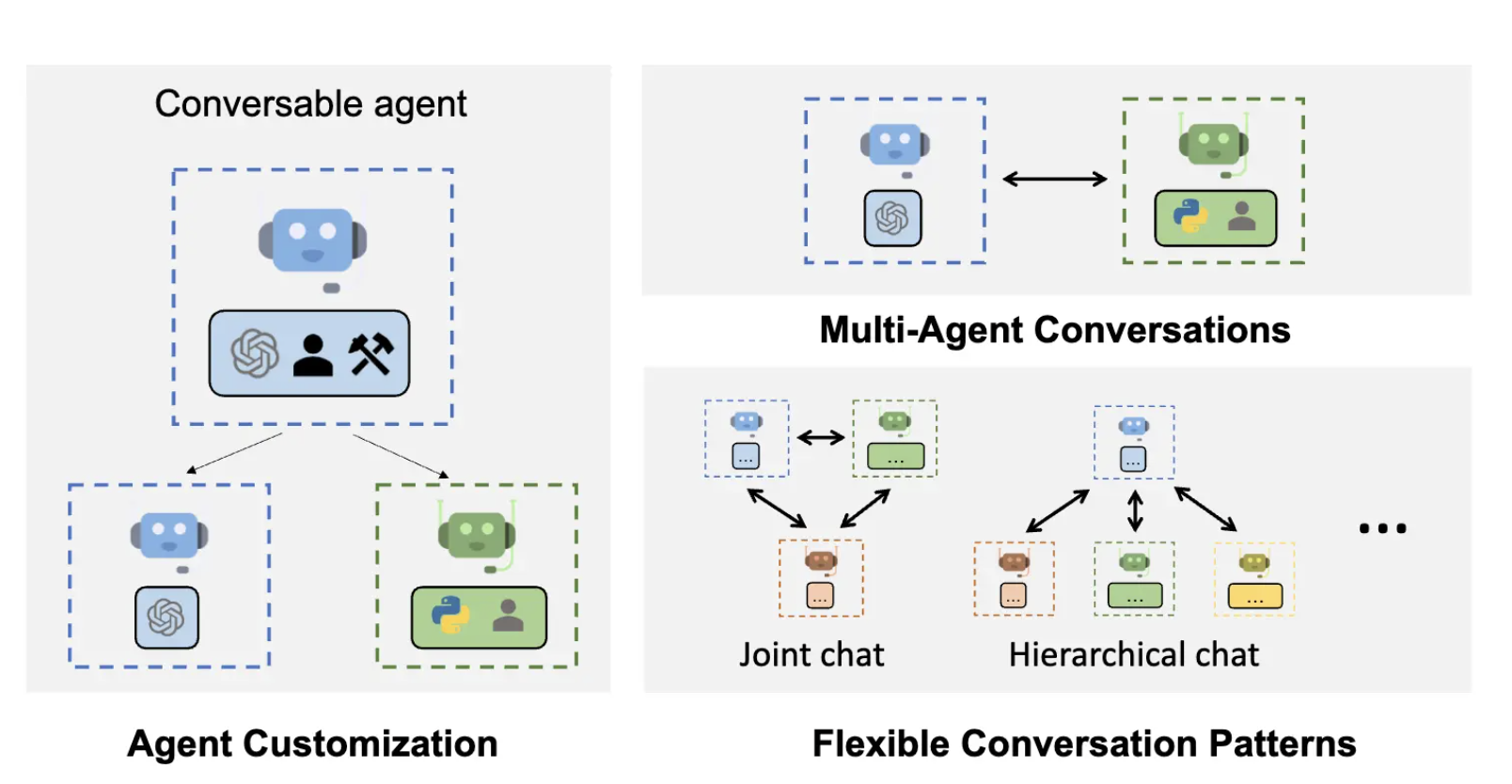


Figure 7: Agents interaction flow