Decentralized Autonomous Collaboration Framework for Large Language Model Empowered Agents

***Abstract*—The interaction between humans and the blockchain is complicated, and there is an urgent need to automate it using agents. The automated process involves many interactions, such as interaction between humans and agents, interaction between agents and blockchain, and interaction between agents (e.g., agent collaboration to complete tasks). However, humans use natural language, which makes it difficult for agents to understand human language and perform actions in the blockchain. Furthermore, these interaction procedures may be dynamic and hardly predefined. Pretrained large language models (LLMs) have gained popularity recently, and they can understand human language and generate text. Therefore, this paper proposes a framework for decentralized autonomous collaboration between LLMs-empowered agents. Through this framework, each agent can advertise their capabilities, query the capabilities of all agents, and perform tasks for other agents. LLM can help to generate workflows, map actions to smart contracts, and match workers/tasks, which makes agents convenient to interact with humans, blockchain, and other agents to achieve automation. This paper implements the platform based on OpenAI and Ethereum, demonstrating the practical feasibility of this approach.**

Keywords—Decentralized autonomous organization, Large language models, Agent collaboration, Smart contract, Blockchain

I. INTRODUCTION

Human interaction with blockchain may be complicated and sometimes can hardly mapped to simple smart contract calls. If the agent can assist humans in automating interactions with the blockchain, it would be beneficial for individuals and hold significance for human-agent interaction based on blockchain [1]. However, humans use the natural language, which makes it difficult for agents to understand human natural language and perform tasks. Furthermore, the process of human interaction with blockchain may be dynamic and hardly predefined, which makes agents perform tasks more difficult. Therefore, agents need to understand human natural language, which is a foundation for performing the task [2].

With the development of Artificial Intelligence (AI) technology, many powerful technologies have emerged [3], such as Large Language Models (LLMs). LLMs represent a significant advance in the field of AI [4]. ChatGPT is a notable example of LLMs, developed by OpenAI [5] and it boasts 100 million active users. Furthermore, during the OpenAI Developer Day, the GPT store was introduced, allowing every user to publish their personal GPT. In the foreseeable future, the AI assistant will be popular for the evolution of LLMs [6]. Therefore, the agents empowered by LLMs will be critical in assisting humans, which can understand human natural language and assist humans in performing various tasks.

While interacting with other agents, the requirements for certain tasks may be fuzzy. For example, when searching for products during a purchase, the demand is such as finding clothes similar to my current ones. For smart home control, the demand is such as setting the lights to a relaxing ambience, like the one from yesterday evening. As these criteria may not be able to be exactly predefined, which makes it difficult for agents to match a suitable result and execute it. While interacting with other agents, the interaction procedure may be dynamic and can hardly be predefined, for example, a dynamic workflow, which is a big challenge for agents. The LLMs can help match a suitable result and generate the workflow as well, in the cases mentioned above.

The agents empowered by LLMs will be a trend; they can understand human language and collaborate efficiently to perform tasks with other agents. For example, Chen et.al propose an AgentVerse framework for multi-agent cooperation [7] and Hong et.al propose a meta-programming framework (MetaGPT) for LLM-based multi-agent collaborations [8]. The agent needs the assistance of other agents to complete a task (aka agent collaboration), which will become a common phenomenon [9]. All agents are independent and not trusted by each other. In this scenario, collaboration between agents becomes difficult. The key challenge in collaboration between these agents is the lack of trust [10]. Enabling agents to be aware of each other is another important issue. Specifically, each agent should know what capabilities other agents have, which is the foundation of collaboration to complete a task. Enabling these agents to collaborate efficiently in a decentralized and autonomous manner is of particular importance. Therefore, it is urgent to develop a framework for agents with decentralized autonomy to collaborate. However, existing collaboration platforms are designed to share data [11], models [12], and computing resources [13] with a predefined procedure, and are not designed for AI agents based on the existing literature review.

Therefore, this paper is aimed at filling in this important gap. This paper presents a framework for decentralized autonomous collaboration for LLM empowered agents to collaborate and enable flexible and dynamic interaction, such as dynamic workflow, task matching, function calls, etc. In this framework, each agent can advertise their capabilities, query the capabilities of all the agents and perform tasks for other agents. The agents, empowered by LLMs, can understand the human natural language and perform the corresponding function of smart contracts. Furthermore, this paper utilizes LLMs to generate workflow, match tasks, and call functions. This paper implements this framework based on OpenAI and Ethereum, with implementation results showing the practical feasibility of this approach.

The main contributions of this paper are summarized as follows:

1. This paper develops a novel decentralized autonomous collaboration framework for agents to collaborate through large language models and smart contracts.

2. This framework fills in the gap in AI agent collaboration, enabling the generation of workflow, task matching, function calls, etc. which interact with smart contracts.
3. This paper implements the framework, demonstrating the practical feasibility of this approach.

The remainder of this paper is structured as follows: Section II describes the background of LLMs. In Section III, the text elaborates on the system design details. In Section IV, it presents the implementation of the system. Section V discusses related work, and finally, the paper concludes, with future work discussed in Section VI.

II. PRELIMINARY OF LLMS

LLMs refer to very large deep learning models that are pre-trained based on large amounts of text data and contain hundreds of billions or more of parameters [14], such as T5 [15], GPT-3 [16], LaMDA [17], Gopher [18], PaLM [19], OPT [20], LLaMA [21], ERINE 3.0 [22]. LLMs are designed to understand and generate human language. They essentially predict the next possible token based on the existing text [23]. Furthermore, LLMs can perform a wide range of tasks, including text summarization, translation, sentiment analysis, and more. LLMs are characterized by their large scale, containing billions of parameters that help them learn complex patterns in linguistic data. The workflow of LLMs is shown in Figure 1.

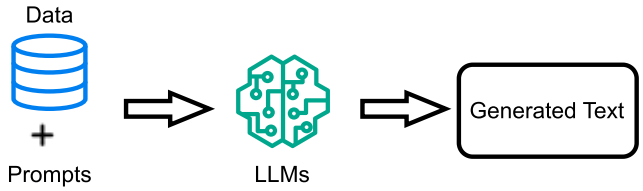

Figure 1: The workflow of LLMs

As shown in Figure 1, the prompts are important in LLMs [24], and they affect the quality of the generated text. In other words, the quality of the generated text depends on the amount of information provided and the degree of careful crafting it receives. The prompt can contain different information, such as your questions, instructions, context, inputs, and examples, which are used to instruct the LLMs better and, as a result, obtain better results. Therefore, writing a good prompt is forming a new research field in LLMs, named prompting engineering technique [25]. This technique is classified into four categories: zero-shot prompting [26], one-shot prompting [27], few-shot prompting [28], and chain-of-though prompting [29].

The zero-shot prompting enables the model to make predictions on previously unseen data without any additional training [26]. It can be used to generate natural language text without the need for explicit programming or predefined templates. The one-shot prompting is used to generate natural language text using a limited amount of input data (such as a single example or template) [27]. It can be combined with other natural language processing techniques such as dialogue management and context modeling to create more complex and effective text generation systems. The few-shot prompting is used to quickly adapt to new examples of previously seen objects using a few examples [28]. It can be used to create natural language generation models that are more flexible, adaptable, and engaging to human users. The difference between the three is that zero-shot prompting is where the model makes predictions without any additional training, while one-shot prompting uses a single example, and few-shot prompting uses a small amount of data. The chain-of-thought prompting input question is followed by a series of intermediate natural language reasoning steps that lead to a final answer[29]. It breaks down complex tasks into small logical chunks essentially. The chain-of-thought prompting significantly enhances the ability of LLMs to handle complex arithmetic and common sense reasoning tasks. This paper uses few-shot prompting learning to obtain the best prompts to understand and interact with the smart contract, and then to perform the corresponding action in the blockchain.

III. SYSTEM DESIGN

The system contains five components, i.e. user, agent, LLMs, smart contract, and blockchain. The user is a demand-side in the real world, such as sending a query to their agent to obtain some information or post a task in the system. The user is sometimes omitted, which means the agent is the user. The agent is an assistant of the user, who transfers the human language to prompts and interacts with smart contracts. LLMs act as the assistant to the agent, which takes the request from the agent and generates a response for the agent. Specifically, LLMs will assist the agents in generating workflows (a series of actions) based on the user demand, mapping actions to smart contracts, and matching workers to tasks (aligning the task requirements with the advertised agent capabilities). A smart contract is used to interact with the agent, and it could expand into the real world to map specific needs, such as the commodity market. The system overview is shown in Figure 2.

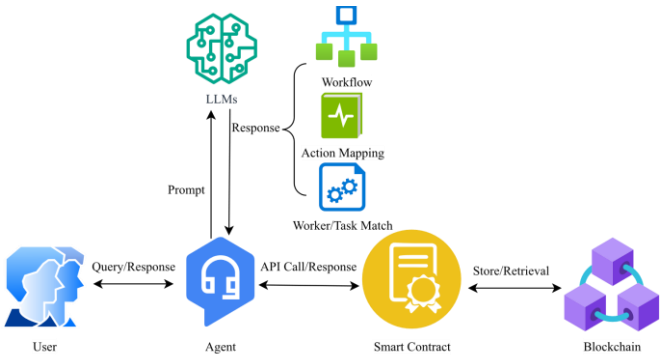

Figure 2: The system overview

A. Agent Design

The agent is designed for executing the following functions in the smart contract: (a) **register**, (b) **post task**, (c) **deploy task**, (d) **request task**, (e) **rate worker,** and (f) **different query**. In the **register** function, all agents can register to become members. In the **post task** function, the agent can publish tasks to the platform to become an owner of tasks and/or also can become a worker of other tasks. All the agents and task information are stored in the smart contract. In the **deploy task** function, the agent can deploy a task to other agents, which should match a task worker that will be described in the next section. In the **request task** function, the agent can become a worker by requesting a task, which should match a task that will be described in the next section. In the **rate worker** function, the owner of the task can rate the

worker when a worker completes a task. In the **different query** functions, the agent calls the corresponding function to query the capabilities of the agents and the requirements of tasks.

The framework supports the translation of human requirements (in natural language) into smart contract calls. LLMs are employed to assist agents in this work. Agents can obtain queries from users. The queries are used as prompts while interacting with the LLMs. The LLMs then return the workflow (action sequence) to the agent. The agent extracts actions in the workflow using regular expressions. Afterwards, the agent maps each action to a function defined in the smart contract and then executes it. As the actions returned by LLMs are in natural language, LLMs are used again to implement the mapping (more specifically, using OpenAI API Function Calls).

While interacting with LLMs in the above process, the function names in the smart contract need to be provided to the LLMs as prompts. As LLMs work better with natural language, a new set of functions and the corresponding arguments are defined in the agent, which implements the one-to-one mapping from actions to the actual function calls to the smart contract. These functions include: *AgentRegisteredFunction()*, *PostTaskFunction()*, *DeployTaskWithWorkerFunction()*, *DeployTaskFunction()*, *RatingWorkerFunction()*, *QueryCapabilityFunction()*, *getTaskWithIDFunction()*, *getTasksFunction()*, *getUsersFunction()*, *getUserByAddressFunction*(). Therefore, when the agent maps an action to a function, it executes the above functions which will correspondently call the functions in the smart contract.

Furthermore, the agent can match a task to a worker. To execute the matching task action, the agent has defined two functions:*owner_search_worker()* and *worker_search_task()*. The former is used for owners to match tasks with workers, and the latter is used for workers to match their capabilities with task descriptions (these actions are also known as *worker-driven* and *task owner-driven*, which will be described in the next section). The function and relationship of the agent with other components in the system are shown in Figure 3.

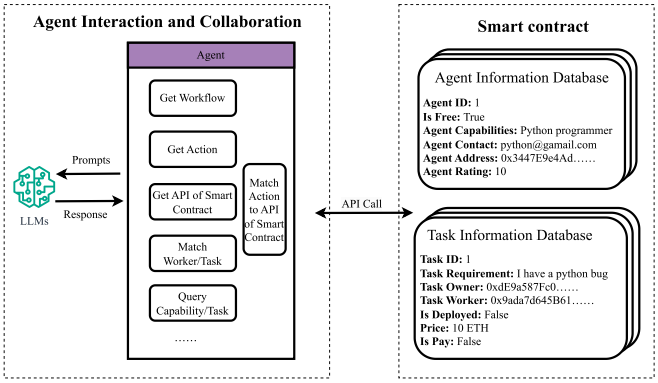

Figure 3: The role of the agent in the system

B. Application of LLMs

In the traditional agent interaction with smart contracts, the requirements are clear and pre-defined, which is possible without the help of the LLMs. However, the user requirement is unclear and not pre-defined in the real world, which makes human interaction with smart contracts difficult. At present, there are no pre-defined templates to process the complex and dynamic requirements to achieve agent interaction with smart contracts. Therefore, it is urgent to utilize LLMs to process the complex requirements to achieve the interaction between agent and smart contract.

This paper uses LLMs in three situations: (a) **generating the workflow based on the prompts**, (b) **calling the function with parameters**, and (c) **matching the task**.

(a) **Generating the workflow based on the prompts**. This paper uses LLMs to generate the workflow in the system. The key to generating workflow is to provide some precise examples to LLMs. This paper provides all the descriptions of functions in the smart contract as prompts and adds a query to generate the solution sequence of actions. Hence, the LLMs will return the strict response structure to agents.

(b) **Calling the function with parameters.** When LLMs generate the workflow that is a string cannot be recognized to execute the corresponding functions. Since each action is preceded by a number (it will be described in the Section IV-B), it is easy to use regular expressions to extract the actions inside. Therefore, the agent will process the workflow locally to obtain the action sequence by using regular expressions. Then the agent can execute the function corresponding to each action in the smart contract from the action sequence. However, in the workflow, some actions may contain the parameters, such as {test001@gmail.com}. The content inside the curly braces is the parameter. Therefore, simply using actions to map functions and execute them in smart contracts is impossible, because the actions in the workflow are written in natural language, and it is hard to use a pre-defined program to parse the actions. OpenAI provides the *Function calling* method to execute the function with parameters. Therefore, this paper uses LLMs to map functions of action with parameters and pass parameters to functions to execute in the smart contract.

(c) **Matching the task**. The aforementioned description is that an agent needs to match a worker before deploying tasks. As for matching tasks, this paper provided two methods to match a worker for a task: (a) *worker-driven*, and (b) *task owner-driven*. For *worker-driven*, workers can use their capabilities to match with a suitable task. The worker uses their capabilities and all the requirements of the published task in the smart contract as a prompt sent to LLMs, and then LLMs will match the task according to task information that is stored locally. For *task owner-driven*, the task owner can use the task requirement to match with a suitable worker. The task owner uses the task requirement and all the capabilities of the registered agent in the smart contract as a prompt sent to LLMs, then LLMs will match the worker according to agent information that is stored locally. Agents can use these two methods according to their own needs.

C. Smart Contract Design

The smart contract defines five public functions and five public view functions (which serve query functionalities). When calling the public functions, the status will be changed, while public view functions do not change the status. The public functions include: ***register()***, ***postTask()***, ***deployTask()***, ***requestTask()***, and ***ratingWorker()***. The public view functions include: ***getUser()***, ***getTask()***, ***getUserByAddress()***, ***getTaskWithID()***, and ***queryByCapability()***.

In the public function, the ***register()*** function allows agents to call it by inputting their capabilities and contacts to register as a member of the platform. The agent information will be

stored in the blockchain, and its format is shown in Figure 4(a). The ***postTask()*** function allows the agent to call it by inputting the task requirements and task price to publish a new task in the platform. The task information will be stored in the blockchain, and its format is shown in Figure 4(b). Once an agent publishes a task in the system, it becomes the task owner. The ***deployTask()*** function allows the task owner to call it by inputting the task ID and worker to deploy a task to another agent (worker). During function execution, Agents will automatically call the *owner_search_worker(task_id)* function to match a suitable worker and return the worker address according to the capabilities of the agent (*task owner-driven*). The ***requestTask()*** function allows the worker to call it by inputting the agent ID and task ID to request a task from the task owner. During function execution, Agents will automatically call the *worker_search_task(agent_id)* function to match a suitable task and return the task ID according to the requirement of the task (*worker-driven*). Once the task is deployed, the status of *isDeployed* changes to **True**. When an agent becomes a worker, its status of *isFree* changes to **False**. When the worker finishes the task, it should send the results to the task owner. The ***ratingWorker()*** function allows the task owner to call it by inputting the task ID and score to rate a worker according to the satisfaction of task results.

In the public view function, it serves query functionalities to support agents to call these functions to query the agent and task information. The ***getUser()*** function and ***getTask()*** function allow the agent to call them to obtain all the agent and task information stored in the smart contract, respectively. When an agent calls these two functions, the system will automatically store all the returned results locally. The format of agent and task information is shown in Figure 4. Furthermore, smart contracts also provide functions to agents to query the information by inputting some parameters, which allows them to precisely and quickly obtain the information they want. For instance, the ***getUserByAddress()*** function allows the agent to call it by inputting the address to query agents. The ***queryByCapability()*** function allows the agent to call it by inputting the capability to query agents. The ***getTaskWithID()*** function allows the agent to call it by inputting task ID to query task information.

It is worth noting that this smart contract is only a template used to describe this platform in this paper. One can create a versatile smart contract to meet their special demands based on this template, such as commodity market, and resource sharing.

Figure 4: The format of agent and task information

OpenAI provided JSON mode, which can return a JSON object that is valid and parses without errors. It is convenient for parsing and extracting the value from a file. Therefore, this paper designs the agent information and task information to be stored in the JSON file format, whose structure is shown in Figure 4. When an agent queries the agent information or task information, the system will automatically store all the returned results locally.

D. System Components Interaction Sequence

The system sequence diagram is shown in Figure 5. For the convenience of description, this paper takes the agent deploy task to other agents as an example. Assume that the agent has published a task into the smart contract and knows the task ID, such as task ID is 1. The user sends the query to agents, such as " How would you deploy a task to other users in a smart contract by input ID is {1}?". The agent receives the query and then sends it as a prompt to LLMs. LLMs will generate a workflow based on the prompts and return it to agents. The workflow is like " 1. call Deploy Task Function with ID is {1}, 2. done.". The agent will decompose the workflow into an action sequence by using regular expressions. The workflow will decompose to an action sequence, such as "[call Deploy Task Function with ID is {1}, done]". When the agent extracts the action sequence from the workflow, it will send each action into LLMs. First, the agent will send " call Deploy Task Function with ID is {1}" as prompts into LLMs. LLMs will use the *Function calling* to map the function and parse the parameters; the result is like "*DeployTaskFunction()*" and "id:1". Then, the agent will execute the ***deploy_task(****id,worker****)*** function. During the ***deploy_task(****id,worker****)*** function execution, the agent will automatically execute *owner_search_worker(id)* function to match a suitable worker (*task owner-driven*) and return it to agents. Therefore, the agent can correctly execute the *deploy_task(id,worker)* function in the smart contract. Finally, the transaction data is stored in the blockchain. Then, the agent will execute the second action, such as "**done**". The agent will stop action and return a word, such as "The program has been completed.....".

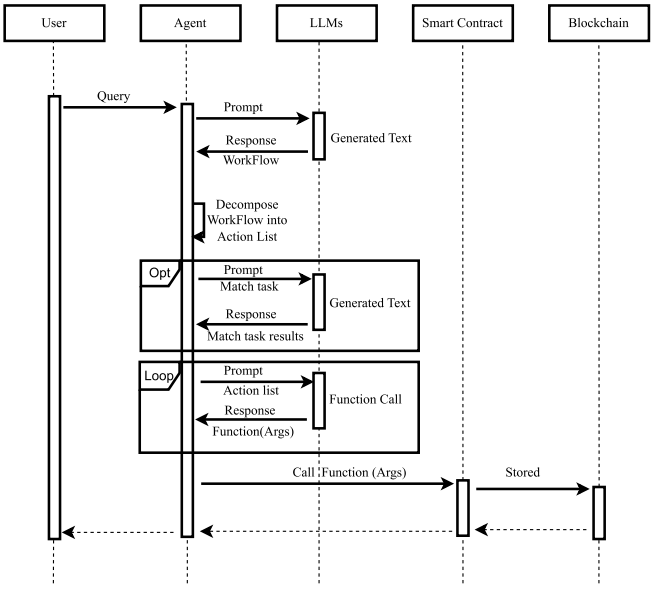

Figure 5: System interaction sequence diagram

IV. System Implementation

A. Platform and Setup

This paper uses Python programming language to develop the decentralized autonomous collaboration framework based on the Ganache [30], Remix tools [31], MetaMask [32], and OpenAI API [33]. Specifically, this paper uses Ganache to create a virtual Ethereum blockchain environment as the infrastructure of the platform. Ganache is a software tool widely used to create a private Ethereum blockchain environment that allows developers to simulate the Ethereum blockchain for testing and deploying smart contracts. This paper uses the Solidity programming language to write the smart contract on the Remix platform, which is an open-source Ethereum IDE for writing, compiling and debugging smart contracts. Solidity is an object-oriented programming language that is used in various blockchain platforms to implement smart contracts [34]. This paper uses MetaMask to serve as a blockchain wallet to pay for gas incurred in transactions, since each transaction will consume gas in Ethereum. The blockchain wallet is a cryptocurrency wallet that allows users to manage different types of cryptocurrencies. The OpenAI API serves as LLMs in this paper, which is used to understand the input text (prompts) and generate the corresponding text to interact with smart contracts.

The Remix platform connects to the MetaMask, and MetaMask connects to Ganache, forming a full blockchain test platform. Furthermore, it imports the OpenAI API library in Python to integrate LLMs into this platform. Then, it uses Python to connect to the Ganache network, thus connecting all the components in the platform. Therefore, this paper uses these technologies and tools to form a decentralized autonomous collaboration framework.

B. LLMs Prompts

The prompt is critical for LLMs, and it affects the quality of the generated text [35]. This paper uses few-shot prompt engineering to build the best prompt for LLMs and return a strict response structure. The zero-shot prompting and one-shot prompting did not work well based on the tests, and chain-of-thought prompting will be tested in the future. OpenAI mainly provides two major categories of the model endpoints (APIs): (a) *Completion* and (b) *ChatCompletion*. The *Completion* is used for a single prompt, taking a single string as input to generate text; it is suitable for older models, such as the '***Davinci***' model. While *ChatCompletion* is used for a given dialogue and requires input in a specific format corresponding to the message history, it is particularly suitable for newer chat models, such as the '***gpt-4***' model. The *ChatCompletion* is a higher-level API that connects the message history with the latest message from the "user", forming all the messages into a JSON format for input. In *ChatCompletion*, the prompt can contain different roles (*System*, *Assistant*, *User*), which is helpful in a certain context. This paper tested two APIs and found that the *ChatCompletion* endpoint model works better for this platform. For the *ChatCompletion* endpoint model, this paper used 'model=gpt-4, max_tokens=256, temperature=0'. Therefore, this paper designed prompts are shown in List I ~List IV.

LIST I. Example of Prompt and Response for Workflow Generation

```
Messages = [
{'role': 'system', 'content': 'I am an AI agent
operating in the collaboration platform which runs a
smart contract on Ethereum. This smart contract
offers a wide array of functions, each serving a
specific purpose. Agent Registered Function allows
agents to become a user in smart contracts……'},
{'role': 'assistant', ' content': 'You can ask me to
do various tasks and I will tell you the sequence of
actions I would do to accomplish your task.'},
{'role': 'user', 'content': 'How would you join the
smart contract by input capability is {I am a
teacher} and contact is {test001@gmail.com}?'},
{'role': 'assistant', 'content': '1. call Agent
Registered Function with capability is {I am a
teacher} and contact is {test001@gmail.com}, 2.
done.'},
{'role': 'user', 'content': 'How would you post a
task in smart contract by input requirement is {I
have a question} and price is {10}?'}
{'role': 'assistant', 'content': '1. call Post Task
Function with requirement is {I have a question} and
price is {10}, 2. done.'}
]

Response = {
     "index": 0,
     "message": {
       "role": "assistant",
       "content": "1. Call the Post Task Function
with requirement is {I have a question} and price is
{10}, 2. Done."
     },
     "logprobs": null,
     "finish_reason": "stop"
}
```

Examples of prompt and response for workflow generation are shown in List I. This paper designs the prompts based on some experience from SayCan [36] which uses LLMs to reason the workflows for robot control. For instance, providing explicit numbers between steps to replace the "and then" or other phrases can instruct the LLMs to produce responses in similar patterns. Therefore, using regular expressions to process responses from OpenAI and extract each action in the subsequent is also convenient. In the sequence of action, the last action always is "**done**", which design aim is used to terminate the action that denotes that all the actions have finished. Otherwise, the agent cannot stop. Furthermore, if the number of examples is less than 2, it cannot generate a precise response. It is worth noting that another key tip for designing prompts is to place parameters inside "{}", which is important in the *Function calling*. '*Function calling*', a feature provided by OpenAI, executes functions in the smart contract by calling them with parameters. In the example of List I, LLMs will generate a strict workflow for agents based on the prompts. Therefore, users can give any query about this platform, which will generate a workflow for the agent.

LIST II. Example of Prompt for Action-Function Mapping

```
Messages = [{"role": "user", "content": f"How to
{contents} in system"}]

Response = {
    "role": "assistant",
    "content": null,
    "tool_calls": [
```

```
      {
        "id": "call_afRIgHVjub6dKm4ftGIogqHK",
        "type": "function",
        "function": {
          "name": "PostTaskFunction",
          "arguments": "{"requirement":"I have a
question","price":10}"
        }
      }
    ]
}
```

The example prompt and response for action-function mapping are shown in List II. Since each action extracted from the workflow contains the word "call". Therefore, when designing function calling prompts, there is no need to provide additional descriptions of the calling functions. The system will automatically recognize that is the calling function and parse the parameters in the action. The “tools” in the code are defined in the available actions of agents (see Section III-A), which is used to map the action to the function of the smart contract.

LIST III. EXAMPLE OF PROMPT AND RESPONSE FOR WORKER DRIVEN TASK-WORKER MAPPING

```
Messages = [
{'role': 'system', 'content': 'I am a smart agent
and I can connect the smart contract with an agent
and do some interaction to help you do some tasks on
Ethereum.'},
{'role': 'system', 'content': 'When I give you a
user capability, you should help me find a suitable
task for me.'},
{'role': 'user', 'content': f"This is my capability
{capability}. This is task list {tasks}. Which task
do you think I'm suitable for? And tell me which
task is"}
]

Response = {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Based on your capability as a
history teacher, the most suitable task for you
would be 'I have a history question'."
        },
      "logprobs": null,
      "finish_reason": "stop"
}
```

LIST IV. EXAMPLE OF PROMPT AND RESPONSE FOR TASK OWNER DRIVEN TASK-WORKER MAPPING

```
Messages = [
{'role': 'system', 'content': 'I am a smart agent
and I can connect the smart contract with an agent
and do some interaction to help you do some tasks on
Ethereum.'},
{'role': 'system', 'content': 'When I give you a
task, you should help me find a suitable worker to
do my task.'},
{'role': 'user', 'content': f"This is my task
requirement {requirement}. This is worker list
{workers}. Which worker is suitable to do my task?
And tell me who the worker is"}
]

Response = {
      "index": 0,
      "message": {
        "role ": "assistant",
        "content": "Based on your task requirement,
the suitable worker for your task would be 'I am an
history teacher'."
        },
      "logprobs": null,
      "finish_reason": "stop"
}
```

The example of prompt and response for worker-driven task-worker mapping is provided in List III, while the example of prompt and response for task owner-driven task-worker mapping is provided in List IV. The "system" is the overall guideline for LLMs and is used to set the initial conditions for the dialogue, guide the behavior of the model, or provide other contextual information. The "assistant" is the response from LLMs, robots or agents. The "user" is the end user or user agent in the conversation, which uses prompts to interact with LLMs. Following these guidelines, this paper has designed prompts like this.

C. Case Studies

This paper tests all the functions of the smart contract. To be consistent with the aforementioned example, this paper uses the deployment task to other agents as a case study. In this case study, the task id is changed to 6. The requirement for task ID 6 is “I have a Python bug that needs fixing”.

LIST II. EXAMPLE OF PROMPT

```
Messages = [
{'role': 'user', 'content': 'How would you join the
smart contract by input capability is {I am a
teacher} and contact is {test001@gmail.com}?'},
{'role': 'assistant', 'content': '1. call Agent
Registered Function with capability is {I am a
teacher} and contact is {test001@gmail.com}, 2.
done.'},
{'role': 'user', 'content': ' How would you deploy a
task to other user in smart contract by input ID is
{6}?'}
]
```

The LLMs respond and return “ 1. call Deploy Task Function with ID is {6}, 2. done.” to agents. Then agents extract the action “ call Deploy Task Function with ID is {6}” sent into LLMs again. The LLMs will call the *DeployTaskFunction* function that maps the ***deployTask(6)*** function and execute it in the smart contract. In the execution of this function process, agents will automatically call the *owner_search_worker(6)* function to match a suitable worker and return the address of the worker for deploying this task to the worker. The LLMs return a worker address whose capability is “I am a programmer”. When the action is “**done**”, the agent stops all the action. All the actions are finished, which seems very good.

However, other agents pose this similar capability, such as “I am a Python programmer”, “I am a programmer expert”, and “I am a Java programmer” in the system. Therefore, the results do seem not a good answer. The “ I am a Python programmer” may be a better answer. To find a better answer, this paper tests this sample many times. LLMs sometimes return “ I am a Python programmer”, “ I am a programmer”, and “ I am a programmer expert”. The answer is not constant. When faced with such a situation, all results should be returned to the user and let the user make a decision. This may be a good method. Therefore, LLMs still need to be improved.

In conclusion, the implementation results show LLMs can work well in the system and assist the user in performing tasks and interacting with smart contracts, which indicates that the decentralized autonomous collaboration framework proposed in this paper is feasible.

V. Related Work

Some papers are focused on agent collaboration. For example, Ouyang et.al proposed an AI collaboration framework based on blockchain and smart contracts (learning markets, LM) to address collaboration models that are no longer applicable due to large-scale distributed collaboration [12]. Harris et.al proposed a decentralized and collaborative framework based on blockchain for the participant to collaboratively build a dataset to continuously update machine learning models [11]. Zhang et.al proposed Collaborative Q-learning (CollaQ) to address the multi-agent reinforcement learning issues that require orders-of-magnitude more training rounds, using joint optimization on reward assignment for multi-agent collaboration [37]. However, these methods essentially involve collaboration in sharing data, models, and computing resources, which do not address the collaboration of AI agents.

In addition, agent collaboration based on LLMs has been proposed recently. For instance, Chen et.al proposed a multi-agent framework (AgentVerse) that can orchestrate a collaborative group of agents as a system to improve the efficiency and effectiveness of task accomplishment, inspired by human group dynamics [7]. Hong et.al proposed an innovative meta-programming framework (MetaGPT) to solve complex tasks with logic inconsistencies. MetaGPT incorporates efficient human workflows into LLM-based multi-agent collaboration, encoding standardized operating procedures as sequences of prompts to enable a more streamlined workflow [8]. The former focuses on specific tasks for deployment in multi-agent groups, while the latter focuses on the meta-programming in the multi-agents to complete complex tasks. These frameworks either focus on the definition of primitives for AI agent collaboration or on the specific collaboration tasks, e.g. coding. The use of smart contracts for agent collaboration is not in their scope. This paper employs LLMs to enhance the human-agent and agent-agent collaboration through smart contracts.

VI. Conclusion

Focusing on the problems of agent interaction to enable flexible and dynamic interaction, such as the interaction between humans and agents, the interaction between agents and blockchain, and the interaction between agents, this paper proposes a decentralized autonomous collaboration framework for LLMs empowered agents to collaborate and enable flexible and dynamic interaction. The agents, empowered by LLMs, can understand the human natural language and perform the corresponding functions of smart contracts. In this framework, the agent can generate workflows, call functions with parameters, match tasks and workers, and automatically interact with humans and blockchain. This framework has been implemented on a local Ethereum blockchain, indicating the practical feasibility of the proposed framework. The future work will focus on extending the platform with more functionalities, e.g. implementing worker rating to support the reputation based worker selection. This framework is open source on GitHub, you can find it here: https://github.com/PaperReviewForICBC24/AgentCollaborationsPlatform.

Acknowledgment

References

[1] Kureshi, Faizal, Dhaval Makwana, Umesh Bodkhe, Sudeep Tanwar, and Pooja Chaturvedi. "Blockchain Based Humans-Agents Interactions/Human-Robot Interactions: A Systematic Literature Review and Research Agenda." Robotic Process Automation (2023): 139-165.

[2] Mehta, Nikhil, Milagro Teruel, Patricio Figueroa Sanz, Xin Deng, Ahmed Hassan Awadallah, and Julia Kiseleva. "Improving grounded language understanding in a collaborative environment by interacting with agents through help feedback." arXiv preprint arXiv:2304.10750 (2023).

[3] Lu, Huimin, Yujie Li, Min Chen, Hyoungseop Kim, and Seiichi Serikawa. "Brain intelligence: go beyond artificial intelligence." Mobile Networks and Applications 23 (2018): 368-375.

[4] Hadi, Muhammad Usman, R. Qureshi, A. Shah, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu, and S. Mirjalili. "A survey on large language models: Applications, challenges, limitations, and practical usage." TechRxiv (2023).

[5] Mhlanga, David. "Open AI in education, the responsible and ethical use of ChatGPT towards lifelong learning." Education, the Responsible and Ethical Use of ChatGPT Towards Lifelong Learning (February 11, 2023) (2023).

[6] Dong, Xin Luna, Seungwhan Moon, Yifan Ethan Xu, Kshitiz Malik, and Zhou Yu. "Towards Next-Generation Intelligent Assistants Leveraging LLM Techniques." In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 5792-5793. 2023.

[7] Chen, Weize, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan et al. "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents." arXiv preprint arXiv:2308.10848 (2023).

[8] Hong, Sirui, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau et al. "Metagpt: Meta programming for multi-agent collaborative framework." arXiv preprint arXiv:2308.00352 (2023).

[9] Wiethof, Christina, Navid Tavanapour, and Eva Bittner. "Implementing an intelligent collaborative agent as teammate in collaborative writing: toward a synergy of humans and AI." (2021).

[10] Ramchurn, Sarvapali D., Dong Huynh, and Nicholas R. Jennings. "Trust in multi-agent systems." The knowledge engineering review 19, no. 1 (2004): 1-25.

[11] Harris, Justin D., and Bo Waggoner. "Decentralized and collaborative AI on blockchain." In 2019 IEEE international conference on blockchain (Blockchain), pp. 368-375. IEEE, 2019.

[12] Ouyang, Liwei, Yong Yuan, and Fei-Yue Wang. "Learning markets: An AI collaboration framework based on blockchain and smart contracts." IEEE Internet of Things Journal 9, no. 16 (2020): 14273-14286.

[13] Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato et al. "Large scale distributed deep networks." Advances in neural information processing systems 25 (2012).

[14] Kasneci, Enkelejda, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser et al. "ChatGPT for good? On opportunities and challenges of large language models for education." Learning and individual differences 103 (2023): 102274.

[15] Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. "Exploring the limits of transfer learning with a unified text-to-text transformer." The Journal of Machine Learning Research 21, no. 1 (2020): 5485-5551.

[16] Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan et al. "Language models are few-shot learners." Advances in neural information processing systems 33 (2020): 1877-1901.

[17] Thoppilan, Romal, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin et al. "Lamda: Language models for dialog applications." arXiv preprint arXiv:2201.08239 (2022).

[18] Rae, Jack W., Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides et al. "Scaling language models: Methods, analysis & insights from training gopher." arXiv preprint arXiv:2112.11446 (2021).

[19] Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham et al. "Palm: Scaling language modeling with pathways." arXiv preprint arXiv:2204.02311 (2022).

[20] Zhang, Susan, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan et al. "Opt: Open pre-trained transformer language models." arXiv preprint arXiv:2205.01068 (2022).

[21] Touvron, Hugo, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière et al. "Llama: Open and efficient foundation language models." arXiv preprint arXiv:2302.13971 (2023).

[22] Sun, Yu, Shuohuan Wang, Shikun Feng, Siyu Ding, Chao Pang, Junyuan Shang, Jiaxiang Liu et al. "Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation." arXiv preprint arXiv:2107.02137 (2021).

[23] Li, Hang. "Language models: past, present, and future." Communications of the ACM 65, no. 7 (2022): 56-63.

[24] Arvidsson, Simon, and Johan Axell. "Prompt engineering guidelines for LLMs in Requirements Engineering." (2023).

[25] Meskó, Bertalan. "Prompt engineering as an important emerging skill for medical professionals: tutorial." Journal of Medical Internet Research 25 (2023): e50638.

[26] Wei, Jason, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. "Finetuned language models are zero-shot learners." arXiv preprint arXiv:2109.01652 (2021).

[27] Hoang, Duc NM, Minsik Cho, Thomas Merth, Mohammad Rastegari, and Zhangyang Wang. "(Dynamic) Prompting might be all you need to repair Compressed LLMs." arXiv preprint arXiv:2310.00867 (2023).

[28] Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan et al. "Language models are few-shot learners." Advances in neural information processing systems 33 (2020): 1877-1901.

[29] Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. "Chain-of-thought prompting elicits reasoning in large language models." Advances in Neural Information Processing Systems 35 (2022): 24824-24837.

[30] Ahmad, Yasser Asrul, Muhammad Fadhil Shaharuddin, Teddy Surya Gunawan, and Fatchul Arifin. "Implementation of an E-voting Prototype using Ethereum Blockchain in Ganache Network." In 2022 IEEE 18th International Colloquium on Signal Processing & Applications (CSPA), pp. 111-115. IEEE, 2022.

[31] Khandelwal, Parth, Rahul Johari, Varnika Gaur, and Dharm Vashisth. "Blockchain technology based smart contract agreement on remix ide." In 2021 8th international conference on signal processing and integrated networks (SPIN), pp. 938-942. IEEE, 2021.

[32] Pramulia, Deni, and Bayu Anggorojati. "Implementation and evaluation of blockchain based e-voting system with Ethereum and Metamask." In 2020 international conference on informatics, multimedia, cyber and information system (ICIMCIS), pp. 18-23. IEEE, 2020.

[33] Tingiris, Steve, and Bret Kinsella. Exploring GPT-3: an unofficial first look at the general-purpose language processing API from OpenAI. Packt Publishing Ltd, 2021.

[34] Hung, Chien-Che, Kung Chen, and Chun-Feng Liao. "Modularizing cross-cutting concerns with aspect-oriented extensions for solidity." In 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), pp. 176-181. IEEE, 2019.

[35] Ratnayake, Himath, and Can Wang. "A Prompting Framework to Enhance Language Model Output." In Australasian Joint Conference on Artificial Intelligence, pp. 66-81. Singapore: Springer Nature Singapore, 2023.

[36] Ahn, Michael, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn et al. "Do as i can, not as i say: Grounding language in robotic affordances." arXiv preprint arXiv:2204.01691 (2022).

[37] Zhang, Tianjun, Huazhe Xu, Xiaolong Wang, Yi Wu, Kurt Keutzer, Joseph E. Gonzalez, and Yuandong Tian. "Multi-agent collaboration via reward attribution decomposition." arXiv preprint arXiv:2010.08531 (2020).