AI VIET NAM – AI COURSE 2023

Multi-agent LLMs

Dinh-Thang Duong, Nguyen-Thuan Duong và Quang-Vinh Dinh

Ngày 5 tháng 5 năm 2024

Phần I: Giới thiệu

Multi-agent LLMs là một hướng phát triển liên quan đến mô hình ngôn ngữ lớn. Trong đó, ta ứng dụng LLMs kết hợp với các **agents** có khả năng sử dụng một công cụ hoặc một chức năng bên ngoài nào đó, nhằm giải quyết bài toán của chúng ta. Ví dụ, ta có thể cài đặt cho LLMs sử dụng calculator để giải quyết các phép tính, từ đó sẽ đảm bảo kết quả đầu ra của mô hình.

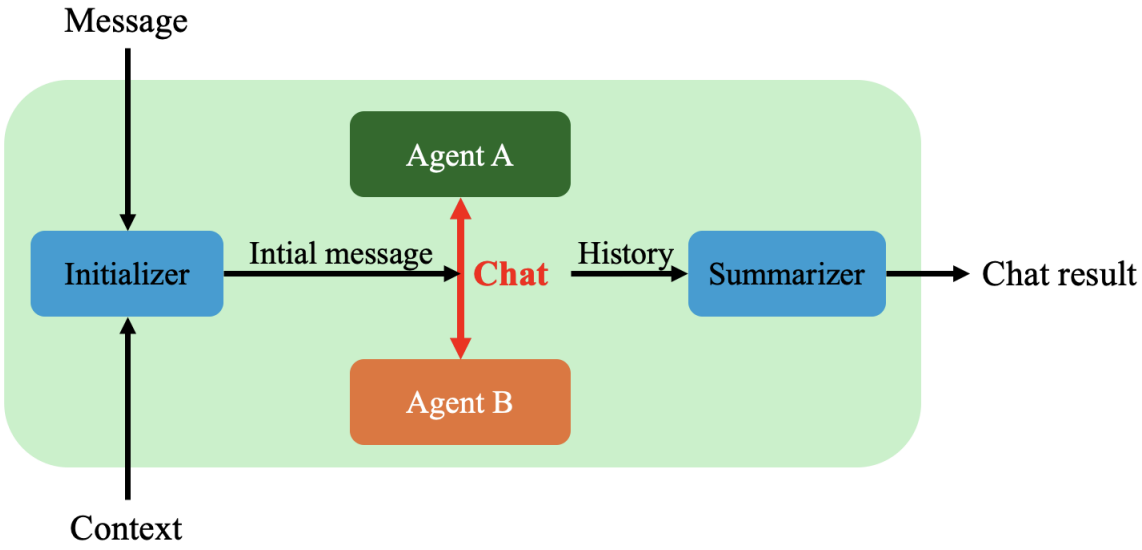
Hình 1: Minh họa về các agent trong môi trường ChatDev. Ảnh: link.

Trong bài viết này, chúng ta sẽ xây dựng một ứng dụng về Multi-agent LLMs tạo sinh code đồng thời kiểm tra code thông qua trình thực thi. Mã nguồn được xây dựng bằng thư viện Autogen. Theo đó, ta sẽ xây dựng một ứng dụng có dạng như Chatbot. Ở đó, sẽ có hai agent tự giao tiếp với nhau gồm AssistantAgent và UserProxyAgent:

- **AssistantAgent** là một LLM, có vai trò tạo sinh code theo yêu cầu và kiểm tra lại code nếu cần.

- **UserProxyAgent** có vai trò gửi yêu cầu tạo sinh đến AssistantAgent và thực thi code nhận được để kiểm tra code đã chạy được hay chưa.

Thông qua hội thoại giữa hai agent này, ta sẽ có được một đoạn code theo yêu cầu đã được kiểm tra thông qua trình thực thi. Tổng quan, pipeline của project như sau:



Hình 2: Pipeline của project.

1. Từ yêu cầu về tạo sinh một đoạn code với chức năng nào đó từ người dùng, ta kết hợp với context có sẵn trong Autogen để hình thành nội dung khởi đầu của cuộc hội thoại giữa hai agent.

2. Với nội dung trên, ta tiến hành cho hai agent giao tiếp với nhau.

3. Từ nội dung chat giữa hai agent, ta trích ra nội dung cần thiết để làm kết quả của chương trình (ở đây là đoạn code được tạo sinh theo yêu cầu và kết quả thực thi).

Phần II: Cài đặt chương trình

Trong phần này, chúng ta sẽ tiến hành cài đặt nội dung của project. Mã nguồn được xây dựng trên hệ điều hành Ubuntu với GPU 24GB. Các bước thực hiện như sau:

1. **Tổ chức thư mục code:** Để mã nguồn trở nên rõ ràng nhằm phục vụ cho mục đích đọc hiểu code, chúng ta sẽ có ảnh minh họa tổ chức cây thư mục như sau:

```
multiagent_llms/
├── agent.py
├── app.py
├── chat.py
├── utils.py
├── OAI_CONFIG_LIST.json
└── requirements.txt
```

2. **Cập nhật file `requirements.txt`:** Ta liệt kê một số thư viện cần thiết để chạy được source code thông qua file requirements như sau:

```
pyautogen==0.2.27
gradio==4.29.0
```

3. **Cập nhật file `agent.py`:** Trong file này, ta định nghĩa các hàm khởi tạo các agents. Nội dung như sau:

   (a) **Import các thư viện và modules cần thiết:**

```python
import autogen
from autogen import AssistantAgent, UserProxyAgent
from autogen.code_utils import extract_code

TIMEOUT = 60
```

   (b) **Khai báo biến config cho LLM (AssistantAgent):** AssistAgent của chúng ta là một LLM. Theo đó, đối với việc sử dụng GPT, AutoGen yêu cầu ta cung cấp một file .json có chứa API KEY và model version. Các bạn tạo một file tên **OAI_CONFIG_LIST.json** có nội dung sau:

```
[
    {
        "model": "gpt-3.5-turbo",
        "api_key": "Type your API_KEY here"
    }
]
```

   Quay lại file code `agent.py`, các bạn đọc file và import config như sau:

```python
config_list = autogen.config_list_from_json("./OAI_CONFIG_LIST.json")
```

(c) **Xây dựng hàm check điều kiện dừng hội thoại cho UserProxyAgent:** UserProxyAgent cần nhận được đoạn code được tạo sinh từ Agent còn lại để thực thi. Vì vậy, ta sẽ khai báo một hàm kết thúc lượt nhắn của UserProxyAgent nếu chưa nhận được code và ngược lại:

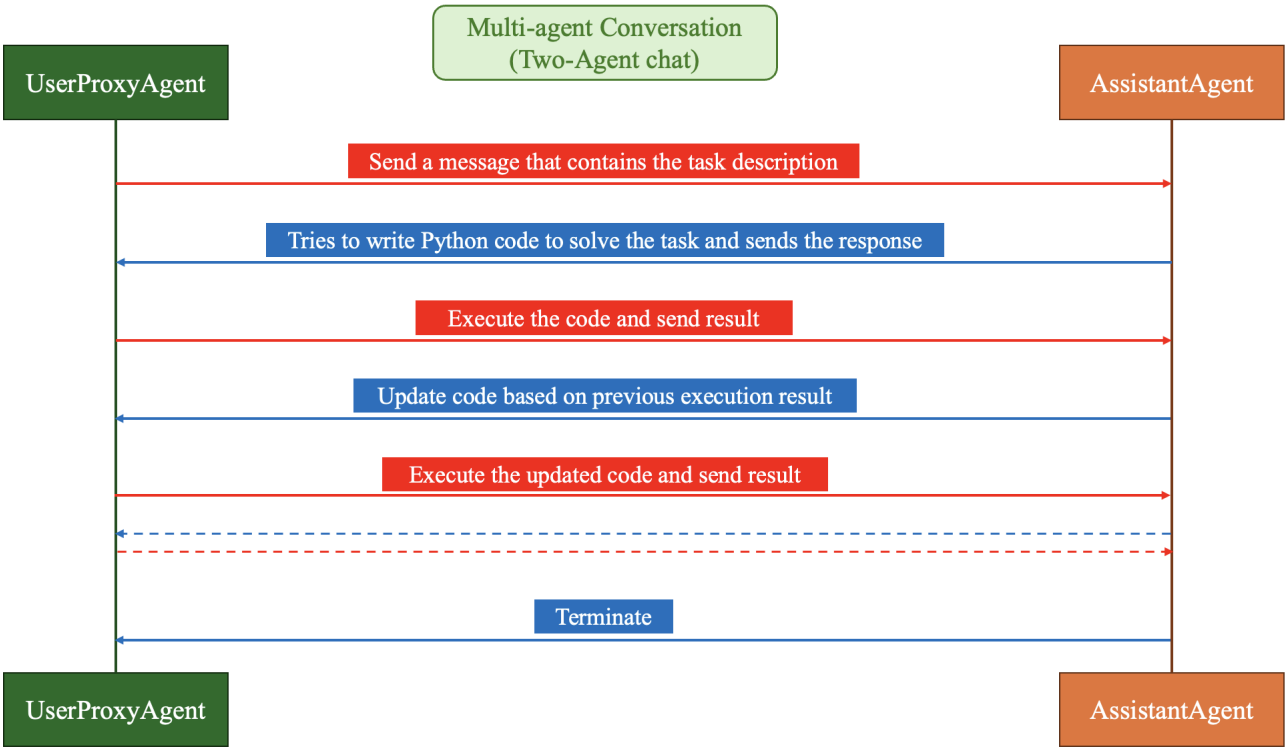
```python
def _is_termination_msg(message):
    if isinstance(message, dict):
        message = message.get("content")
        if message is None:
            return False
    cb = extract_code(message)
    contain_code = False
    for c in cb:
        if c[0] == "python":
            contain_code = True
            break
    return not contain_code
```

(d) **Xây dựng hàm khởi tạo các agents:** Để thuận tiện trong việc triển khai code, ta xây dựng hàm `initialize_agents()` để tạo hai agent, sử dụng trong phần code của file khác trong source code:

```python
def initialize_agents():
    assistant = AssistantAgent(
        name="assistant",
        max_consecutive_auto_reply=5,
        llm_config={
            "timeout": TIMEOUT,
            "config_list": config_list,
        },
    )

    userproxy = UserProxyAgent(
        name="userproxy",
        human_input_mode="NEVER",
        is_termination_msg=_is_termination_msg,
        max_consecutive_auto_reply=5,
        code_execution_config={
            "work_dir": "coding",
            "use_docker": False,
        },
    )

    return assistant, userproxy
```



Hình 3: Minh họa về tương tác giữa hai agent - UserProxyAgent (Trình thực thi code) và AssistantAgent (LLM tạo sinh code).

4. **Cập nhật file** `utils.py`**:** Trong file này, ta định một class liên quan đến threading, sẽ được sử dụng ở file `chat.py`. Nội dung file `utils.py` như sau:

```python
import sys
import threading

LOG_LEVEL = "INFO"
TIMEOUT = 60

class thread_with_trace(threading.Thread):
    def __init__(self, *args, **keywords):
        threading.Thread.__init__(self, *args, **keywords)
        self.killed = False
        self._return = None

    def start(self):
        self.__run_backup = self.run
        self.run = self.__run
        threading.Thread.start(self)

    def __run(self):
        sys.settrace(self.globaltrace)
        self.__run_backup()
        self.run = self.__run_backup

    def run(self):
        if self._target is not None:
            self._return = self._target(*self._args, **self._kwargs)

```

```
    def globaltrace(self, frame, event, arg):
        if event == "call":
            return self.localtrace
        else:
            return None

    def localtrace(self, frame, event, arg):
        if self.killed:
            if event == "line":
                raise SystemExit()
        return self.localtrace

    def kill(self):
        self.killed = True

    def join(self, timeout=0):
        threading.Thread.join(self, timeout)
        return self._return
```

5. **Cập nhật file** `chat.py`**:** Trong file này, ta định nghĩa các hàm xây dựng hộp hội thoại (chatbox) giữa hai agent. Từ đó, cho chúng tương tác qua lại và lấy về kết quả cuối cùng - lịch sử cuộc trò chuyện giữa hai agent, từ đó trích xuất được đoạn code và kết quả thực thi. Theo đó, ta triển khai như sau:

   (a) **Import các thư viện và module cần thiết:**

```
import os
from autogen import OpenAIWrapper
from agent import initialize_agents, config_list
from utils import *
```

   (b) **Khởi tạo agents:** Ta khởi tạo hai agent, UserProxyAgent và AssistantAgent. Đây là hai agent sẽ tương tác qua lại với nhau trong chatbox:

```
assistant, userproxy = initialize_agents()
```

   (c) **Xây dựng hàm chuyển đổi format lịch sử hội thoại:** Để đưa thông tin các cuộc hội thoại giữa 2 agent tại quá khứ trong chatbox vào model GPT, ta sẽ xây dựng hàm chuyển đổi thông tin lịch sử ta ghi nhận được sang format của OpenAI thông qua hàm `chat_to_oai_message()`:

```
def chat_to_oai_message(chat_history):
    """Convert chat history to OpenAI message format."""
    messages = []
    if LOG_LEVEL == "DEBUG":
        print(f"chat_to_oai_message: {chat_history}")
    for msg in chat_history:
        messages.append(
            {
                "content": msg[0].split()[0] if msg[0].startswith("exitcode") else msg[0],
                "role": "user",
            }
        )
        messages.append({"content": msg[1], "role": "assistant"})
    return messages
```

   (d) **Xây dựng hàm chuyển đổi ngược format lịch sử hội thoại:** Để hiển thị thông tin lịch sử lên giao diện của chúng ta sau khi chạy xong, ta cần lấy lại thông tin lịch sử từ GPT. Vì thông tin đang ở format của OpenAI, ta cần có hàm chuyển đổi ngược lại thành một list như ban đầu. Theo đó, ta khai báo hàm `oai_message_to_chat()`:

```python
def oai_message_to_chat(oai_messages, sender):
    """Convert OpenAI message format to chat history."""
    chat_history = []
    messages = oai_messages[sender]
    if LOG_LEVEL == "DEBUG":
        print(f"oai_message_to_chat: {messages}")
    for i in range(0, len(messages), 2):
        chat_history.append(
            [
                messages[i]["content"],
                messages[i + 1]["content"] if i + 1 < len(messages) else "",
            ]
        )
    return chat_history
```

(e) **Xây dựng hàm khởi tạo đoạn hội thoại ban đầu:** Khi bắt đầu chương trình, ta cần khởi tạo môi trường chat cho hai agent. Ta có hàm `initiate_chat()` như sau:

```python
def initiate_chat(config_list, user_message, chat_history):
    if LOG_LEVEL == "DEBUG":
        print(f"chat_history_init: {chat_history}")

    if len(config_list[0].get("api_key", "")) < 2:
        chat_history.append(
            [
                user_message,
            ]
        )
        return chat_history
    else:
        llm_config = {
            "timeout": TIMEOUT,
            "config_list": config_list,
        }
        assistant.llm_config.update(llm_config)
        assistant.client = OpenAIWrapper(**assistant.llm_config)

    if user_message.strip().lower().startswith("show file:"):
        filename = user_message.strip().lower().replace("show file:", "").strip()
        filepath = os.path.join("coding", filename)
        if os.path.exists(filepath):
            chat_history.append([user_message, (filepath,)])
        else:
            chat_history.append([user_message, f"File {filename} not found."])
        return chat_history

    assistant.reset()
    oai_messages = chat_to_oai_message(chat_history)
    assistant._oai_system_message_origin = assistant._oai_system_message.copy()
    assistant._oai_system_message += oai_messages

    try:
        userproxy.initiate_chat(assistant, message=user_message)
        messages = userproxy.chat_messages
        chat_history += oai_message_to_chat(messages, assistant)
    except Exception as e:
        chat_history.append([user_message, str(e)])

```

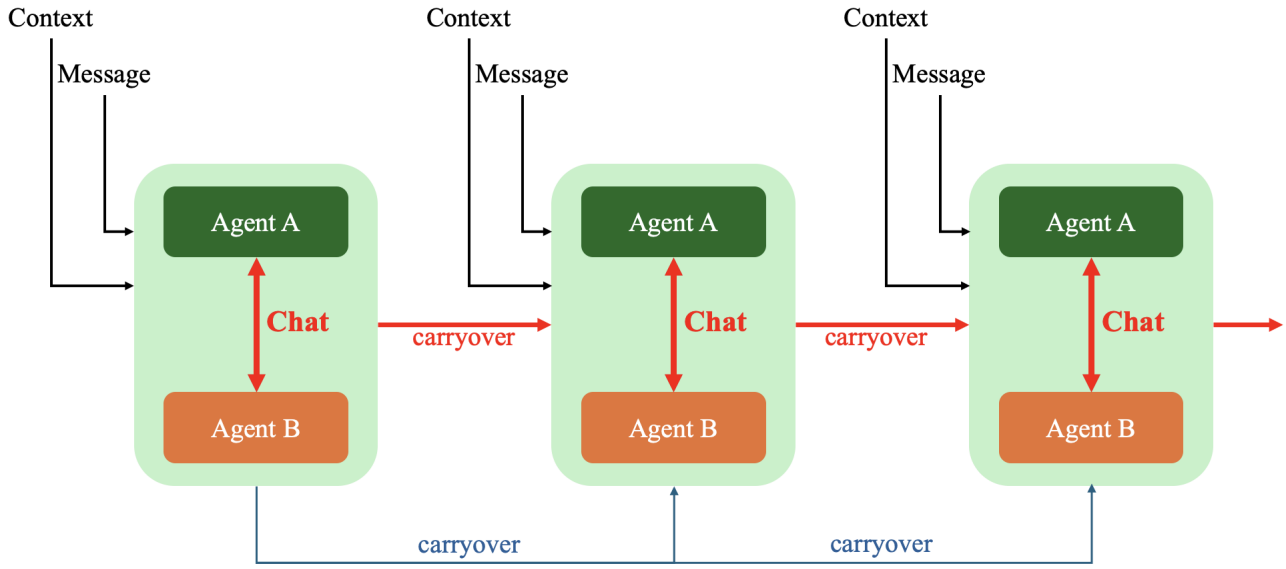
```
    assistant._oai_system_message = assistant._oai_system_message_origin.copy()
    if LOG_LEVEL == "DEBUG":
        print(f"chat_history: {chat_history}")
    return chat_history
```

(f) **Xây dựng hàm trả về kết quả đoạn hội thoại:** Ta xây dựng hàm trả về kết quả của chương trình. Kết quả chương trình của chúng ta là lịch sử đoạn hội thoại giữa hai agent (code của AssistantAgent và kết quả thực thi code của UserProxyAgent). Theo đo, hàm trả về kết quả được triển khai như sau:

```
def chatbot_reply_thread(input_text, chat_history, config_list):
    """Chat with the agent through terminal."""
    thread = thread_with_trace(target=initiate_chat, args=(config_list, input_text, chat_history))
    thread.start()
    try:
        messages = thread.join(timeout=TIMEOUT)
        if thread.is_alive():
            thread.kill()
            thread.join()
            messages = [
                input_text,
                "Timeout Error: Please check your API keys and try again later.",
            ]
    except Exception as e:
        messages = [
            [
                input_text,
                str(e) if len(str(e)) > 0 else "Invalid Request to OpenAI, please check your API keys.",
            ]
        ]
    return messages

def chatbot_reply(input_text, chat_history, config_list):
    """Chat with the agent through terminal."""
    return chatbot_reply_thread(input_text, chat_history, config_list)

def chat_respond(message, chat_history, model, oai_key, aoai_key, aoai_base):
    chat_history[:] = chatbot_reply(message, chat_history, config_list)
    if LOG_LEVEL == "DEBUG":
        print(f"return chat_history: {chat_history}")
    return ""
```



Hình 4: Minh họa việc trả về nội dung đoạn hội thoại (carryover) giữa hai agent. Các carryover theo thời gian trở thành lịch sử đoạn hội thoại làm context cho các lần chat khác.

6. **Cập nhật file** `app.py`**:** Cuối cùng, ta xây dựng một giao diện đơn giản để tương tác với chương trình, trực quan hóa hội thoại giữa hai agent cũng như kết quả của chương trình mà ta mong muốn. Theo đó, ta sẽ xây dựng giao diện bằng thư viện Gradio. Triển khai như sau:

   (a) **Import các thư viện và modules cần thiết:** Tại đây, ta sẽ import thư viện Gradio. Ta sẽ sử dụng giao diện chat cơ bản của Gradio là `ChatInterface`:

```python
import gradio as gr
from gradio import ChatInterface

from chat import chat_respond

LOG_LEVEL = "INFO"
```

   (b) **Xây dựng block Gradio:** Để triển khai giao diện, ta xây dựng một block gradio với các thành phần trên giao diện như sau:

```python
with gr.Blocks() as demo:

    description = gr.Markdown("""
        # Microsoft AutoGen
        ## Multi-agent Conversation
        """)

    chatbot = gr.Chatbot(
        [],
        elem_id="chatbot",
        bubble_full_width=False,
        avatar_images=(
            "../images/human.png",
            "../images/autogen.png",
        ),
        render=False,
        height=600,
```

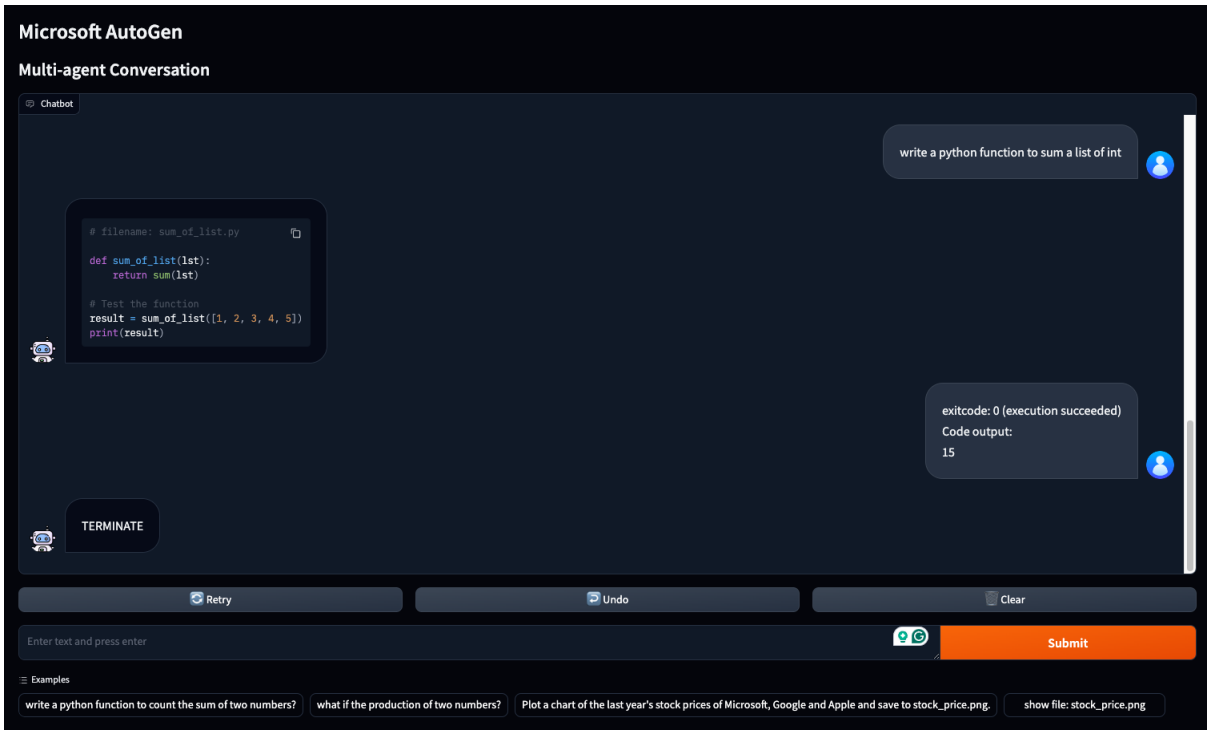
```
    )

    txt_input = gr.Textbox(
        scale=4,
        show_label=False,
        placeholder="Enter text and press enter",
        container=False,
        render=False,
        autofocus=True,
    )

    chatiface = ChatInterface(
        chat_respond,
        chatbot=chatbot,
        textbox=txt_input,
        examples=[
            ["write a python function to count the sum of two numbers?"],
            ["what if the production of two numbers?"],
            ["Plot a chart of the last year's stock prices of Microsoft, Google and Apple and save to stock_price.png."],
            ["show file: stock_price.png"],
        ],
    )
```

(c) **Khởi động giao diện:** Cuối cùng, ta cài đặt lệnh khởi động giao diện khi chạy file app.py như sau:

```
if __name__ == "__main__":
    demo.launch(share=False, server_name="0.0.0.0", server_port=7868)
```



Hình 5: Giao diện ứng dụng Multi-agent LLMs sau khi được triển khai và chạy thử một ví dụ.

Phần III: Câu hỏi trắc nghiệm

1. Trong Multi-agent LLMs, khái niệm agent có thể được hiểu như là?
   (*a*) Là một Docker Container.
   (*b*) Là một máy ảo (Virtual machine).
   (*c*) Là một mô hình LLM.
   (*d*) Là một thực thể (Entity) có khả năng nhận và gửi thông điệp (Message) đến các agent khác.

2. Đâu là ưu điểm của việc sử dụng phối hợp của nhiều agent (Multi-agent)?
   (*a*) Giải quyết một vấn đề nhanh hơn.
   (*b*) Có khả năng giải quyết đa dạng các bài toán hơn.
   (*c*) Giải quyết được các vấn đề, bài toán khó và phức tạp.
   (*d*) Dễ dàng cài đặt.

3. Xét đoạn code dưới đây:

```python
import os
from autogen import ConversableAgent

agent = ConversableAgent(
    "chatbot",
    llm_config={"config_list": [{
                        "model": "gpt-3.5-turbo",
                        "api_key": os.environ.get("OPENAI_API_KEY")
                    }]},
)
```

   Agent trên được khởi tạo như là:
   (*a*) Một mô hình LLM.
   (*b*) Một công cụ thực thi code.
   (*c*) Một máy ảo.
   (*d*) Công cụ lưu trữ lịch sử hội thoại.

4. Một agent có khả năng thực thi script (Code Executors) (ví dụ như python code) sẽ tồn tại rủi ro tiềm ẩn nào?
   (*a*) Giống như chúng ta chạy code python nên không có rủi ro.
   (*b*) LLM có thể sinh ra các script nhằm đánh cắp dữ liệu của máy HOST.
   (*c*) Khiến máy HOST bị chậm do tiêu tốn tài nguyên CPU/GPU.
   (*d*) Tiêu tốn không gian lưu trữ dữ liệu mà agent tải về.

5. Đâu **KHÔNG** là một giải pháp để giảm thiểu rủi ro cho một Code Executors Agent?
   (*a*) Luôn luôn yêu cầu sự kiểm tra của con người trước khi cho agent thực thi code.
   (*b*) Sử dụng máy ảo để thực thi code.
   (*c*) Sử dụng công cụ Containerization (ví dụ Docker Container).
   (*d*) Sử dụng Jupyter Notebook để thực thi code.

6. Xét đoạn code dưới đây:

```
executor = DockerCommandLineCodeExecutor(
    image="python:3.11-slim",
    timeout=10,
    work_dir=temp_dir.name,
)
```

Ý nghĩa của dòng thứ 2 là gì?

(*a*) Khai báo tên của docker container.

(*b*) Khai báo tên của docker image.

(*c*) Khai báo tên docker image được dùng để thực thi.

(*d*) Khai báo tên của Code Executor Agent.

7. Xét đoạn code dưới đây:

```
human_proxy = ConversableAgent(
    "human_proxy",
    human_input_mode="ALWAYS",
)
```

Ý nghĩa của dòng thứ 3 là gì?

(*a*) Là message từ user truyển cho agent tại thời điểm khởi tạo.

(*b*) Cho phép agent có khả năng thực thi code.

(*c*) Luôn luôn yêu cầu message của con người trước khi agent thực thi.

(*d*) Cho phép con người truyển message bất cứ khi nào.

8. Trong Autogen framework, đâu là cách để chấm dứt cuộc hội thoại, trò chuyện?

(*a*) Dùng **max_turns** parameter trong hàm **initiate_chat**.

(*b*) Dùng **max_consecutive_auto_reply** parameter khi khởi tạo agent.

(*c*) Dùng **is_termination_msg** parameter khi khởi tạo agent.

(*d*) Cả 3 cách trên.

9. Xét đoạn code dưới đây:

```
executor = DockerCommandLineCodeExecutor(
    image="python:3.11-slim",
    timeout=10,
    work_dir=temp_dir.name,
)

userproxy = UserProxyAgent(
    name="userproxy",
    human_input_mode="NEVER",
    is_termination_msg=lambda msg: "TERMINATE" in msg["content"],
    max_consecutive_auto_reply=5,
    code_execution_config={"executor": executor},
)

executor.stop()
```

Ý nghĩa của line thứ 15 là gì?

(*a*) Chấm dứt cuộc hội thoại.

(*b*) Ngắt hội thoại của **userproxy** agent.

(*c*) Dừng docker container được dùng để thực thi code.

(*d*) Dừng việc thực thi code.

10. Vai trò của **Group Chat Manager** trong group chat là gì?

(*a*) Chọn ra một Agent duy nhất để thực hiện task.

(*b*) Nhận task từ Boss Agent, sau đó phân tán task cho tất cả các Agent trong group.

(*c*) Thu thập kết quả từ các Agent sau đó đưa ra câu trả lời.

(*d*) Nhận task từ Boss Agent, chọn một Agent thực hiện task, lặp lại cho tới khi hoàn thành.

- Hết -