AI VIET NAM
@aivietnam.edu.vn

AI VIET NAM – AI COURSE 2025

Tutorial: Agent2Agent (A2A) Protocol

Hồ Quang Hiển Nguyễn Quốc Thái Đinh Quang Vinh

I. Giới thiệu

Hình 1: Giới thiệu A2A Protocol.

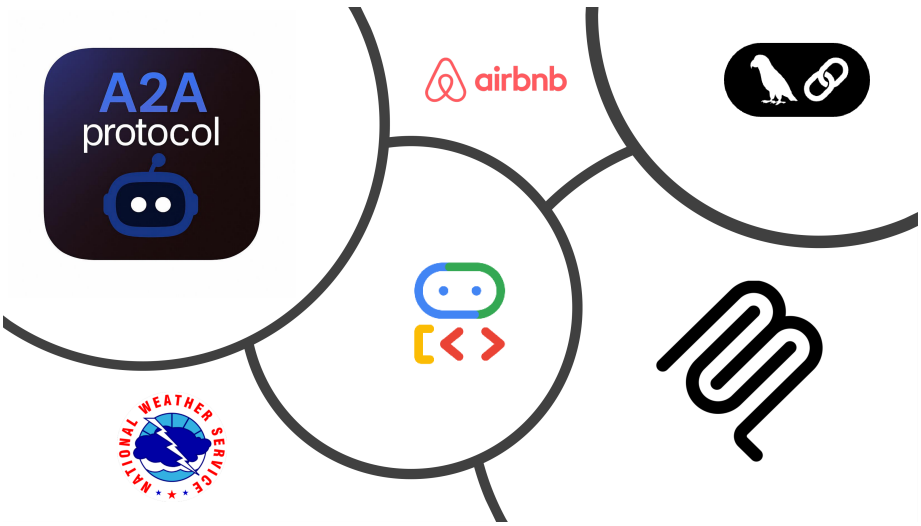
Sự hợp tác giữa các Agent độc lập, được xây dựng trên các nền tảng công nghệ khác nhau, là một thách thức lớn trong môi trường doanh nghiệp hiện đại. Hãy hình dung một quy trình làm việc phức tạp như sau:

- **Agent A** – Chatbot chăm sóc khách hàng, được xây dựng bằng *LangGraph*, phát hiện ra một sự cố kỹ thuật nghiêm trọng từ phía người dùng.
- Agent A cần chuyển tiếp sự cố này cho **Agent B** – một agent chẩn đoán chuyên biệt, được phát triển nội bộ bằng *CrewAI*. Agent B có khả năng phân tích log hệ thống, xác định nguyên nhân và đề xuất hướng xử lý.
- Sau khi phân tích, Agent B xác định rằng cần triển khai một bản vá phần mềm cụ thể để khắc phục lỗi.
- Cuối cùng, Agent B phải chuyển nhiệm vụ triển khai này cho **Agent C** — một agent bên thứ ba, được xây dựng bằng *LlamaIndex*, có chức năng truy cập và cập nhật trực tiếp vào môi trường triển khai của khách hàng.

Không có một tiêu chuẩn giao tiếp chung, việc tích hợp ba agent này là một quá trình tùy chỉnh phức tạp và thiếu tính ổn định. Các lập trình viên sẽ phải tạo các lớp chuyển đổi riêng cho từng cặp agent. Bất kỳ thay đổi nào về thành phần agent hoặc mỗi lần cập nhật API đều có thể dẫn đến lỗi dây chuyền, ảnh hưởng đến tính ổn định của quy trình hợp tác liên agent. Kết quả là một hệ thống thiếu tính linh hoạt, dễ bị phân tán chức năng, gây cản trở cho việc mở rộng quy mô hoặc thích ứng với yêu cầu nghiệp vụ mới. Để hiện thực hóa khả năng hợp tác hiệu quả giữa các Agent, hệ thống cần một cơ chế chung cho phép các Agent có thể:

- **Khám phá năng lực (Capability Discovery):** Mỗi Agent phải công bố rõ các tác vụ mình hỗ trợ, định dạng dữ liệu chấp nhận và phương thức giao tiếp để các Agent khác có thể đánh giá và định tuyến phù hợp.
- **Thỏa thuận cách tương tác (UX Negotiation):** Các Agent cần thống nhất về hình thức trao đổi thông tin (văn bản, biểu mẫu, file, stream...) để đảm bảo hiểu đúng và phối hợp hiệu quả.
- **Quản lý tác vụ và trạng thái (Task and State Management):** Mỗi tác vụ phải được theo dõi rõ ràng trong suốt vòng đời (submitted → working → completed), kể cả khi qua nhiều Agent xử lý.
- **Hợp tác an toàn (Secure Collaboration):** Giao tiếp giữa các Agent phải đảm bảo xác thực, phân quyền và bảo mật, đặc biệt trong các hệ thống phân tán hoặc xuyên tổ chức.

Trước thực trạng đó, cần một cách tiếp cận toàn diện để thiết lập một cơ chế giao tiếp chung, cho phép các Agent dị nền tảng có thể tương tác hiệu quả, linh hoạt và an toàn. Và giải pháp nổi bật đang được đề xuất ở đây chính là: **Agent-to-Agent (A2A) Protocol**.

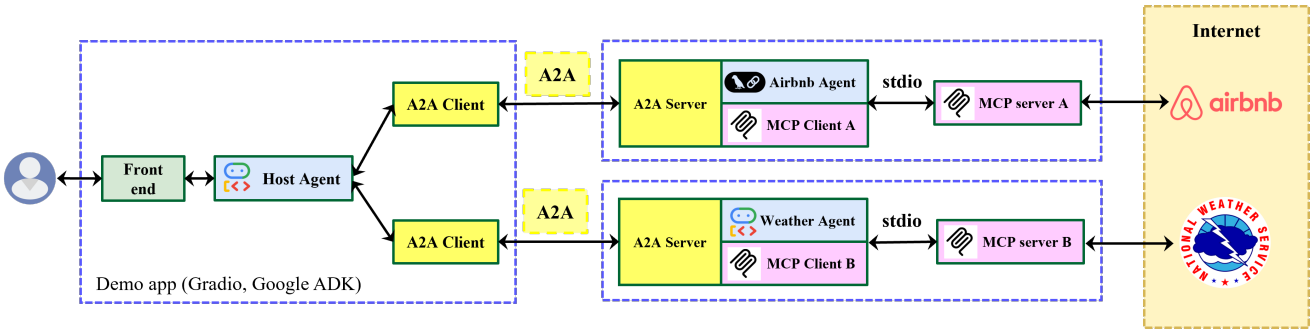


Hình 2: Sự kết hợp của nhiều Agent là xu hướng tất yếu.

Agent-to-Agent (A2A) Protocol là một giao thức tiêu chuẩn, được thiết kế nhằm giải quyết toàn diện các thách thức trên trong môi trường AI đa agent. A2A định nghĩa một tập hợp quy tắc giao tiếp và định dạng thông điệp chuẩn hóa, đóng vai trò như lớp trừu tượng (abstraction

layer) giữa các Agent – giúp che giấu các chi tiết về ngôn ngữ lập trình, framework, hoặc cách triển khai nội bộ của từng Agent cụ thể.

Nhờ đó, các Agent có thể liên kết, phối hợp và hoán đổi linh hoạt, bất kể sự khác biệt về công nghệ nền tảng. Việc này tăng cường khả năng tương tác liên nền tảng, thúc đẩy khả năng mở rộng, và tạo tiền đề cho sự phát triển của các hệ thống AI hợp tác phức tạp trong môi trường doanh nghiệp hiện đại.



Hình 3: Demo sử dụng A2A và MCP.

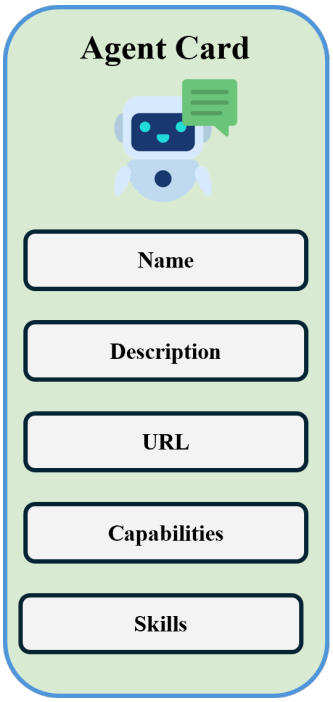
Mục lục

II. Các khái niệm cốt lõi của A2A

II.1. Agent Card

Agent Card là một tệp JSON công khai đóng vai trò như "danh thiếp kỹ thuật số"của một Agent tuân thủ giao thức A2A. Nó chứa các thông tin mô tả về Agent và là một tệp JSON tiêu chuẩn, thường được lưu trữ tại đường dẫn `(/.well-known/agent.json)`

Mục đích: Phục vụ cho việc khám phá Agent (Agent Discovery). Các client sẽ truy xuất tệp này để tìm hiểu năng lực và phương thức tương tác của agent.



Hình 4: Minh họa Agent Card đơn giản

Các trường chính:

- `name` **(bắt buộc)**: Tên dễ hiểu, hiển thị của agent. Dùng để phân biệt trong giao diện hoặc log hệ thống.
- `description` **(tùy chọn)**: Mô tả ngắn gọn vai trò hoặc chức năng chính của agent.
- `url` **(bắt buộc)**: Endpoint HTTP(S) nơi agent lắng nghe và xử lý các lệnh A2A theo giao thức JSON-RPC. Thường là: `https://.../a2a`.
- `version` **(bắt buộc)**: Phiên bản hiện tại của agent hoặc phiên bản chuẩn A2A mà agent tuân thủ (ví dụ: `"1.0.0"`).
- `capabilities` **(bắt buộc)**: Các năng lực giao thức mà agent hỗ trợ, bao gồm:

  - `streaming` (boolean, mặc định: `false`): Agent có hỗ trợ phản hồi dạng streaming qua `tasks/sendSubscribe` với SSE (Server-Sent Events) hay không.
  - `pushNotifications` (boolean, mặc định: `false`): Agent có khả năng gửi thông báo chủ động đến client qua webhook (khi client đã đăng ký trước) hay không.
  - `stateTransitionHistory` (boolean, mặc định: `false`): Agent có lưu và trả về lịch sử chuyển trạng thái của các Task hay không.

- `authentication` **(tùy chọn)**: Mô tả các phương thức xác thực được agent yêu cầu trước khi cho phép tương tác:
  - `schemes` **(bắt buộc nếu có authentication)**: Danh sách các phương thức xác thực được hỗ trợ như: `"bearer"`, `"apiKey"`, hoặc định dạng tùy chỉnh.
  - `credentials` **(tùy chọn)**: Thông tin hoặc URL cung cấp gợi ý về cách lấy token/API key (không nên dùng cho thông tin nhạy cảm).

- `defaultInputModes` / `defaultOutputModes` **(tùy chọn)**: Mô tả kiểu dữ liệu mà agent mặc định chấp nhận (đầu vào) hoặc tạo ra (đầu ra). Ví dụ: `"text"`, `"application/json"`, `"image/png"`.

- `skills` **(bắt buộc)**: Danh sách các kỹ năng hoặc chức năng mà agent cung cấp. Mỗi phần tử là một đối tượng gồm:
  - `id` **(bắt buộc)**: Định danh duy nhất (string) cho skill — dùng trong giao tiếp hoặc định tuyến task.
  - `name` **(bắt buộc)**: Tên dễ đọc cho skill (hiển thị với người dùng hoặc agent khác).
  - `description` **(tùy chọn)**: Mô tả ngắn về mục đích của skill.
  - `inputModes` / `outputModes` **(tùy chọn)**: Các loại dữ liệu mà skill chấp nhận đầu vào và tạo ra đầu ra — có thể khác với mặc định của agent.
  - `examples` **(tùy chọn)**: Các ví dụ minh họa cho prompt hoặc use case cụ thể — giúp client hiểu và sử dụng đúng skill.

Ví dụ về cấu trúc Agent Card (mang tính minh hoạ):

```
{
  "name": "Image Generation Agent",
  "description": "Generates images based on text prompts.",
  "url": "https://api.example-image-agent.com/a2a",
  "version": "1.0.0",
  "capabilities": {
    "streaming": true,
    "pushNotifications": false,
    "stateTransitionHistory": true
  },
  "authentication": {
    "schemes": ["apiKey"]
  },
  "defaultInputModes": ["text"],
```

```
  "defaultOutputModes": ["image/png"],
  "skills": [
    {
      "id": "generate_image",
      "name": "Generate Image",
      "description": "Creates an image from a textual description.",
      "inputModes": ["text"],
      "outputModes": ["image/png"],
      "examples": ["Generate an image of a 'blue cat wearing a top hat'"]
    }
  ]
}
```

II.2. Giao thức JSON-RPC 2.0

A2A sử dụng giao thức **JSON-RPC 2.0** làm nền tảng chuẩn để các Agent có thể giao tiếp một cách thống nhất, không phụ thuộc vào nền tảng. Giao thức này cho phép client gửi yêu cầu, theo dõi tiến trình, nhận phản hồi hoặc hủy bỏ các tác vụ (task) đang xử lý. Dưới đây là danh sách các phương thức JSON-RPC chính mà một A2A Server cần hỗ trợ:

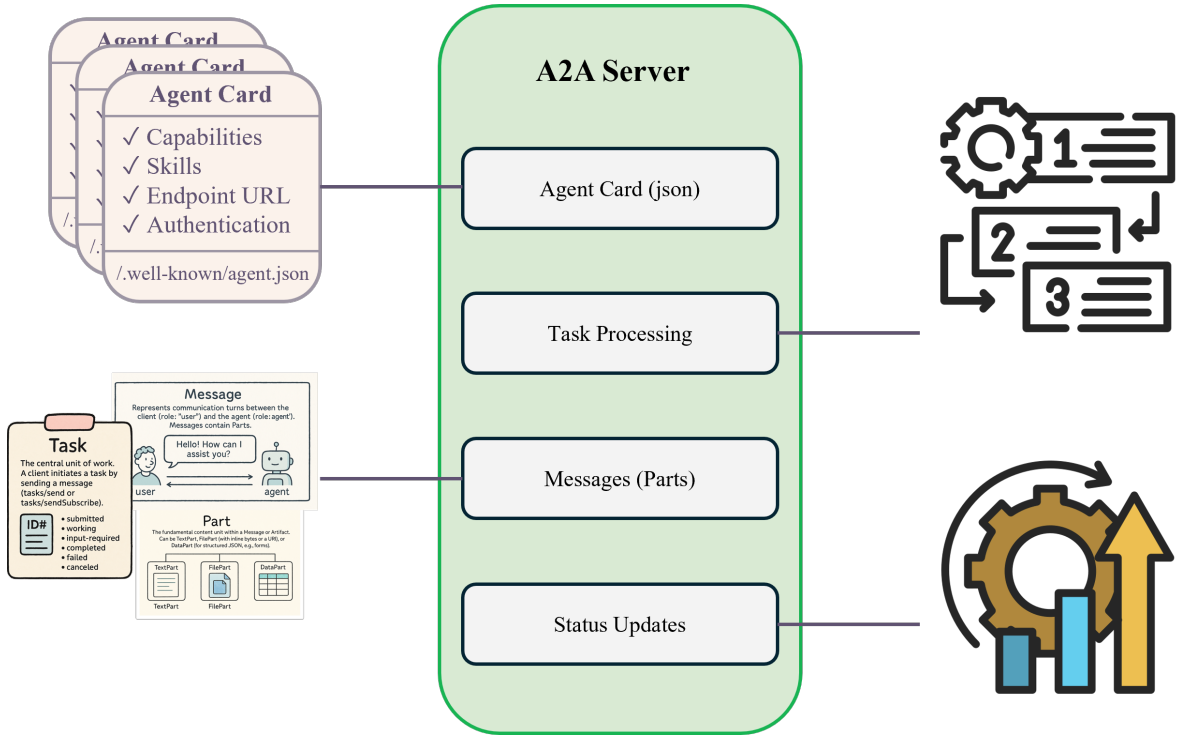
- `tasks/send`: Gửi một task mới và nhận phản hồi khi hoàn tất.
- `tasks/sendSubscribe`: Gửi task và nhận tiến trình xử lý theo thời gian thực qua SSE.
- `tasks/get`: Truy vấn trạng thái hiện tại của một task cụ thể.
- `tasks/cancel`: Hủy một task đang xử lý (nếu có thể).
- `tasks/pushNotification/set`: Cấu hình webhook để nhận thông báo chủ động từ agent.
- `tasks/pushNotification/get`: Xem lại webhook đã đăng ký.
- `tasks/resubscribe`: Khôi phục luồng SSE nếu bị ngắt kết nối.

Ví dụ minh họa cách sử dụng `tasks/send` trong thực tế:

```
{ "jsonrpc": "2.0",
  "method": "tasks/send",
  "params": {
    "taskId": "abc-123-task",
    "message": {
      "role": "user",
      "parts": [
        { "text": "Hãy tạo một hình ảnh con mèo đang cưỡi rồng." }
      ]
    }
  },
  "id": "req-001"}
```

II.3. A2A Server

A2A Server là thành phần được triển khai kèm theo mỗi Agent chuyên biệt (ví dụ: LangGraph, CrewAI, LlamaIndex, ...), đảm nhiệm vai trò tiếp nhận và xử lý các tác vụ được gửi đến từ A2A Client thông qua giao thức **JSON-RPC**.



Hình 5: Các bước hoạt động của A2A Server trong hệ thống agent

A2A Server chịu trách nhiệm chính cho các hoạt động sau:

- Xử lý các phương thức JSON-RPC như:
  - `tasks/send`
  - `tasks/get`
  - `tasks/cancel`
- Quản lý vòng đời tác vụ (`Task Lifecycle`) từ:
  - `submitted` → `working` → `completed` / `failed`
- Gửi phản hồi đến client qua các cơ chế:
  - HTTP response (phản hồi đồng bộ)
  - SSE (Server-Sent Events) – để truyền dữ liệu theo thời gian thực
  - Webhook – để gửi thông báo chủ động đến client

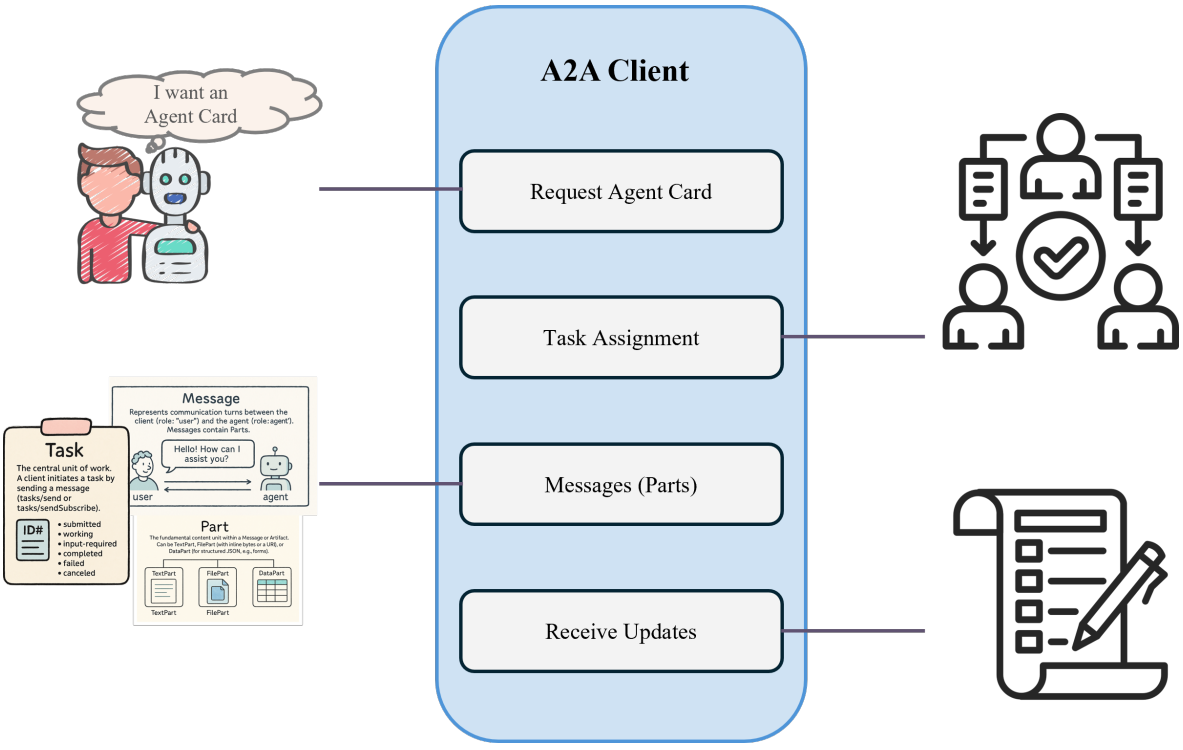
- Liên kết trực tiếp với logic xử lý nội bộ của Agent, ví dụ:
  - Phân tích log hệ thống
  - Tổng hợp dữ liệu
  - Triển khai bản vá hoặc cập nhật hệ thống

II.4. A2A Client

A2A Client là thành phần trung gian thực hiện điều phối tác vụ (task) đến các A2A Server đã đăng ký, thông qua giao thức chuẩn **JSON-RPC 2.0**.

Thực hiện:

- Gửi các lệnh JSON-RPC như: `tasks/send`, `tasks/sendSubscribe`.
- Lắng nghe phản hồi từ Server qua:
  - HTTP response
  - SSE (stream kết quả)
  - Webhook (callback chủ động từ Agent)
- Tái kết nối hoặc đăng ký lại stream khi bị gián đoạn (`resubscribe`).
- Không cần biết công nghệ nội bộ của Agent – chỉ cần đọc Agent Card và điều phối Agent thông qua giao thức A2A.



Hình 6: Các bước hoạt động của A2A Client trong hệ thống agent

II.5. Task

Task là một yêu cầu công việc do client khởi tạo, được agent xử lý. Giao tiếp trong A2A luôn xoay quanh **Task**.

Vòng đời Task (Task Lifecycle):

- `submitted`: Client vừa gửi yêu cầu.
- `working`: Agent đang xử lý.
- `input-required`: Agent cần thêm thông tin từ client.
- `completed`: Xử lý thành công.
- `failed`: Lỗi khi xử lý.
- `canceled`: Client chủ động hủy.
- `unknown`: Không xác định.

Thông tin chính trong một Task:

- `id`: Mã định danh duy nhất (thường là UUID).
- `sessionId` (tuỳ chọn): Dùng để nhóm các task liên quan.
- `status`: Trạng thái hiện tại, kèm thời gian và message mới nhất.
- `artifacts` (tuỳ chọn): Kết quả tạo ra (file, dữ liệu, ảnh...).
- `history` (tuỳ chọn): Lưu lịch sử hội thoại theo lượt.
- `metadata` (tuỳ chọn): Thông tin phụ dạng key-value.

Ví dụ về Giao tiếp dựa trên Task – (Request):

```
{
 "jsonrpc": "2.0",
 "method": "tasks/send",
 "params": {
   "taskId": "20250615123456",
   "message": {
     "role": "user",
     "parts": [
       {
         "text": "Find flights from New York to Miami on 2025-06-15"
       }
     ]
   }
 },
```

```
  "id": "12345"
}
```

Ví dụ về Giao tiếp dựa trên Task – (Response)

```
{
 "jsonrpc": "2.0",
 "id": "12345",
 "result": {
   "taskId": "20250615123456",
   "state": "completed",
   "messages": [
     {
       "role": "user",
       "parts": [
         {
           "text": "Find flights from New York to Miami on 2025-06-15"
         }
       ]
     },
     {
       "role": "agent",
       "parts": [
         {
           "text": "I found the following flights from New York to Miami
           on June 15, 2025:\n\n1. Delta Airlines DL1234: Departs JFK 08:00,...
         }
       ]
     }
   ],
   "artifacts": []
 }
}
```

II.6. Message

Đại diện cho một lượt hội thoại trong quá trình xử lý một Task.

- **Cấu trúc (theo `types.py`):**
  - `role` (enum: "user"hoặc "agent"): Xác định ai là người gửi.
  - `parts` (bắt buộc): Danh sách các phần nội dung (xem mục Part bên dưới).
  - `metadata` (tuỳ chọn): Thông tin bổ sung đi kèm thông điệp.

II.7. Part

Là đơn vị nội dung cơ bản trong một Message hoặc Artifact. Một Message có thể chứa nhiều Part với các kiểu khác nhau.

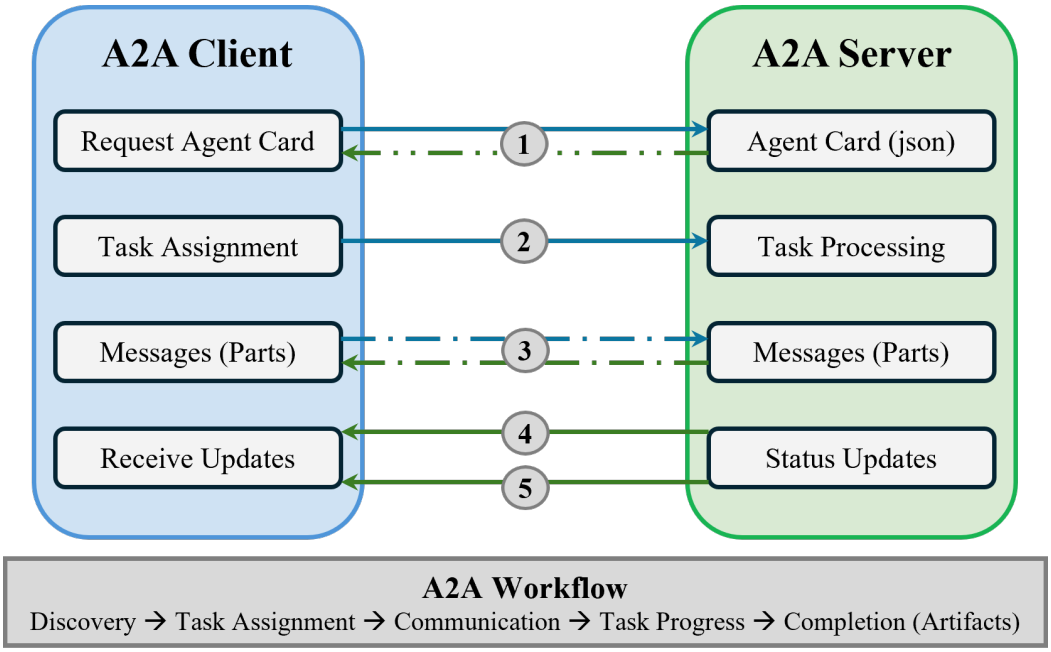
- **Các loại Part (theo `types.py`):**
  - `TextPart`: Chứa văn bản thuần (`text`).
  - `FilePart`: Đại diện cho tệp tin, có thể gồm:
    - `bytes`: Nội dung tệp mã hóa Base64 (dùng cho tệp nhỏ)
    - `uri`: Đường dẫn đến tệp (có thể kèm `name`, `mimeType`)
  - `DataPart`: Dữ liệu JSON có cấu trúc (`data`), dùng cho biểu mẫu, kết quả dạng bảng, v.v.
- Mỗi Part có thể đính kèm `metadata` (tuỳ chọn).

II.8. Artifact

Đại diện cho kết quả do agent tạo ra trong quá trình thực hiện task, khác biệt với hội thoại (Message). Ví dụ: mã nguồn, ảnh, tài liệu, hoặc dữ liệu có cấu trúc.

- **Cấu trúc (theo `types.py`):**
  - `name` / `description` (tuỳ chọn): Tên và mô tả artifact
  - `parts` (bắt buộc): Danh sách nội dung (giống như trong Message)
  - `metadata` (tuỳ chọn): Thông tin đi kèm
  - `index`, `append`, `lastChunk` (tuỳ chọn): Dùng khi streaming artifact lớn thành nhiều phần nhỏ

II.9. Mô hình tương tác giữa 2 Agent



Hình 7: Mô hình tương tác giữa 2 Agent.

Một phiên giao tiếp tiêu chuẩn giữa hai agent trong giao thức Agent2Agent (A2A) thường trải qua 5 giai đoạn chính sau:

1. **Khám phá năng lực (Discovery)**:
   - **Mục tiêu:** Giúp Client Agent hiểu được Remote Agent có thể làm gì trước khi gửi task.
   - **Cách thức:** Client Agent thực hiện HTTP GET tới endpoint `/.well-known/agent.json`.
   - **Kết quả:** Nhận về *Agent Card* chứa thông tin định danh, năng lực tác vụ, endpoint API, định dạng dữ liệu hỗ trợ và cơ chế xác thực.
2. **Giao nhiệm vụ (Task Assignment)**:
   - **Mục tiêu:** Khởi tạo một task mới để Remote Agent xử lý.
   - **Cách thức:** Client gửi yêu cầu thông qua phương thức `tasks/send` hoặc `tasks/sendSubscribe` (nếu cần nhận phản hồi theo dạng streaming).
   - **Nội dung:** Task bao gồm tên tác vụ, dữ liệu đầu vào, mô tả yêu cầu và các cấu hình liên quan.
   - **Chuẩn giao tiếp:** Giao tiếp tuân theo JSON-RPC 2.0.
3. **Trao đổi dữ liệu (Communication)**:

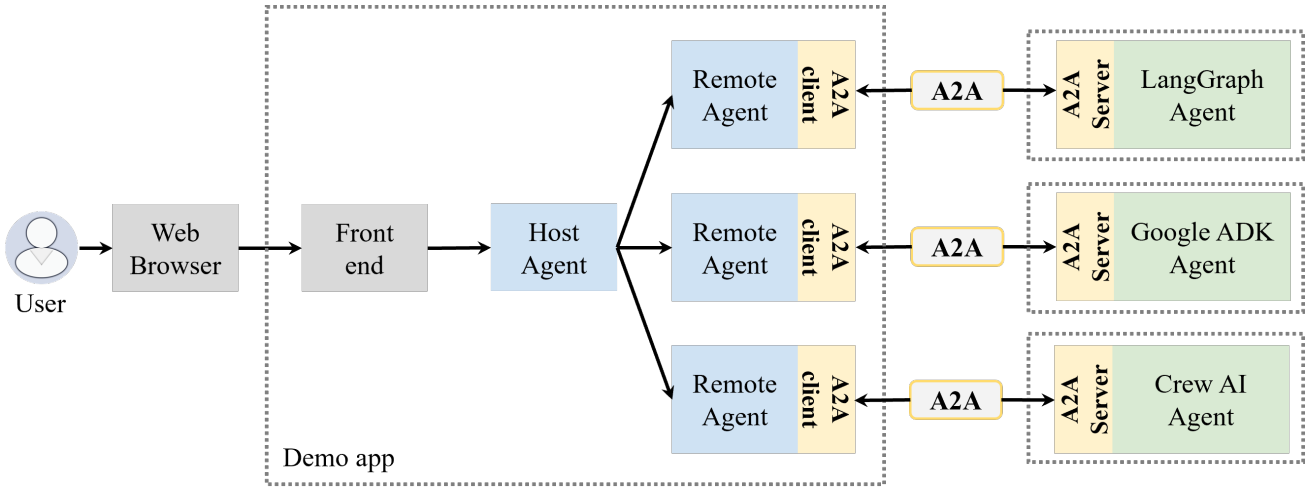
- **Mục tiêu:** Hỗ trợ truyền tải dữ liệu bổ sung hoặc tương tác hai chiều trong quá trình xử lý task.
- **Cách thức:** Gửi các phần dữ liệu nhỏ (gọi là *Message Part*) qua lại giữa hai Agent.
- **Dữ liệu hỗ trợ:** văn bản, biểu mẫu có cấu trúc (form), file đính kèm, URI tham chiếu...

4. **Cập nhật tiến trình (Task Progress)**:
   - **Mục tiêu:** Cho phép Client theo dõi trạng thái của task theo thời gian thực.
   - **Phương pháp:** Remote Agent gửi các cập nhật định kỳ hoặc theo sự kiện về trạng thái tác vụ (submitted, working, completed, failed).
   - **Cơ chế truyền:** Thông qua SSE (Server-Sent Events) hoặc webhook (push).
5. **Hoàn tất và trả kết quả (Completion)**:
   - **Mục tiêu:** Cung cấp đầu ra cuối cùng cho Client sau khi task hoàn thành.
   - **Dữ liệu trả về:** Có thể là văn bản, tệp, dữ liệu JSON hoặc URI đến artifact.
   - **Cách truyền:** Gửi kèm theo cập nhật trạng thái cuối, hoặc thông qua cơ chế pull (nếu chỉ định URI).

Luồng này phản ánh đầy đủ vòng đời của một tác vụ trong hệ sinh thái A2A, theo trình tự:

Discovery → *Task Assignment* → *Communication* → *Task Progress* → *Completion*

II.10. Mô hình phối hợp nhiều Agent



Hình 8: Mô hình phối hợp nhiều Agent.

Hình trên minh họa kiến trúc một hệ thống đa agent (multi-agent system) sử dụng giao thức A2A để giao tiếp và điều phối tác vụ giữa các Agent khác nhau. Quá trình này có thể chia thành hai quy trình chính:

Quy trình 1: Khởi tạo và khám phá Agent chuyên biệt

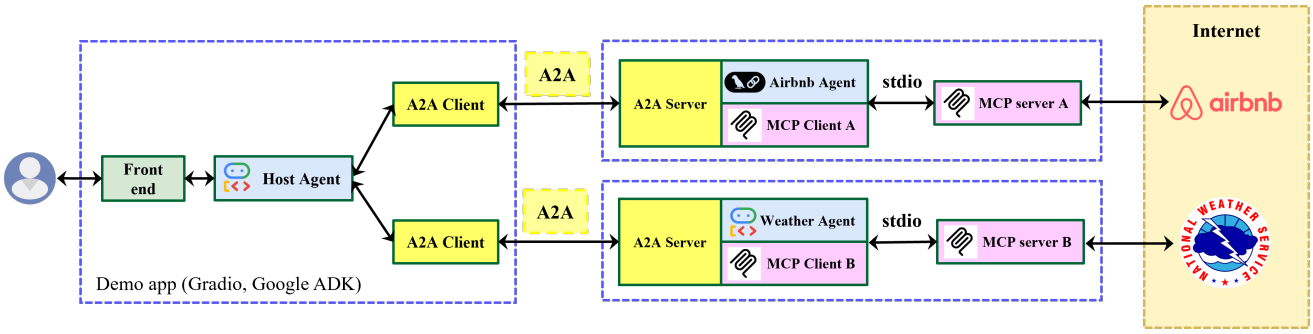
- Mỗi Agent chuyên biệt (được xây dựng trên LangGraph, CrewAI, Google ADK, v.v.) hoạt động như một **A2A Server**, và công bố năng lực của mình thông qua `/.well-known/agent.json` (Agent Card).
- Host Agent chứa các **A2A Client** , và các client này chịu trách nhiệm kết nối đến các A2A Server của các agent chuyên biệt (như Weather Agent, Airbnb Agent).
- A2A Client sẽ truy xuất Agent Card để biết các khả năng, endpoint và định dạng giao tiếp mà Agent Server hỗ trợ.
- Kết quả: Hệ thống sẵn sàng phối hợp với nhiều Agent không đồng nhất, mà không cần viết tay từng lớp tích hợp riêng biệt.

Quy trình 2: Từ yêu cầu người dùng đến phản hồi cuối cùng

- Người dùng nhập yêu cầu từ giao diện web, yêu cầu này được gửi về Host Agent.
- Host Agent phân tích yêu cầu và chọn Remote Agent phù hợp, dựa trên thông tin đã khám phá từ Agent Card.
- Remote Agent gửi task đến Agent Server qua các phương thức như `tasks/send` hoặc `tasks/sendSubscribe`.
- Trong quá trình xử lý, Agent Server sẽ cập nhật tiến trình và kết quả qua luồng SSE hoặc webhook.
- Host Agent tổng hợp dữ liệu đầu ra và gửi phản hồi cuối cùng về frontend cho người dùng.

Lưu ý: Khái niệm "Remote Agent" trong một số tài liệu trình bày nằm ở phía client (đại diện cho các agent đã kết nối trong một host Agent), trong khi tài liệu khác đặt nó phía server. Tuy nhiên, bản chất kiến trúc của A2A là sự tương tác giữa hai vai trò: **A2A Client** và **A2A Server** dù cho Remote Agent nằm ở vị trí nào.

III. Xây dựng mô hình áp dụng A2A và MCP



Hình 9: Mô hình áp dụng A2A và MCP

III.1. Tổng quan về mô hình

Hệ thống được thiết kế theo kiến trúc **đa Agent phân tán** (distributed multi-agent), trong đó các thành phần Agent tương tác thông qua **A2A Protocol** dựa trên chuẩn JSON-RPC over HTTP, cho phép phân tách rõ ràng giữa điều phối và xử lý tác vụ. Trung tâm hệ thống là

Routing Agent (Host Agent), chịu trách nhiệm tiếp nhận truy vấn từ người dùng (thông qua giao diện Gradio), phân tích mục đích truy vấn và định tuyến tác vụ đến các Remote Agent chuyên biệt tương ứng.

Routing Agent không trực tiếp thực hiện truy vấn, mà chỉ đóng vai trò điều phối luồng tác vụ. Sau khi phân tích nội dung câu hỏi (bằng mô hình ngôn ngữ - LLM), agent này khởi tạo một message task và gửi đến Remote Agent tương ứng thông qua **A2A Client**. Việc gửi nhận được thực hiện qua endpoint HTTP do Remote Agent cung cấp.

Weather Agent là một Remote Agent chuyên xử lý các truy vấn liên quan đến điều kiện khí tượng. Khi nhận một truy vấn qua **A2A Server**, tác tử này đưa truy vấn vào mô hình ngôn ngữ để phân tích ngữ nghĩa và xác định các tham số cần thiết (ví dụ: tên thành phố, khung thời gian). Dựa trên đó, mô hình ra quyết định gọi công cụ từ MCP toolset. Công cụ này sẽ thực hiện một API call thực tế đến dịch vụ weather.gov (hệ thống thời tiết quốc gia Hoa Kỳ) – thông qua giao thức HTTP. Dữ liệu trả về dạng JSON sẽ được phân tích, tóm tắt và gửi ngược lại Host Agent thông qua A2A. (Lưu ý: do giới hạn API, Weather Agent hiện chỉ hỗ trợ địa điểm trong lãnh thổ Hoa Kỳ.)

Airbnb Agent xử lý các truy vấn liên quan đến chỗ ở – chẳng hạn: "Tìm homestay ở Đà Lạt tuần này."Sau khi nhận truy vấn từ Host Agent thông qua A2A, tác tử này sử dụng mô hình ngôn ngữ để phân tích yêu cầu, trích xuất các tham số như vị trí địa lý, thời gian lưu trú, loại hình chỗ ở, v.v. Sau đó, agent gọi công cụ phù hợp từ MCP toolset. Tool này truy xuất dữ liệu từ dịch vụ Airbnb thông qua API chính thức. Kết quả được phân tích, lọc theo tiêu chí phù hợp và phản hồi lại Host Agent thông qua giao thức A2A.

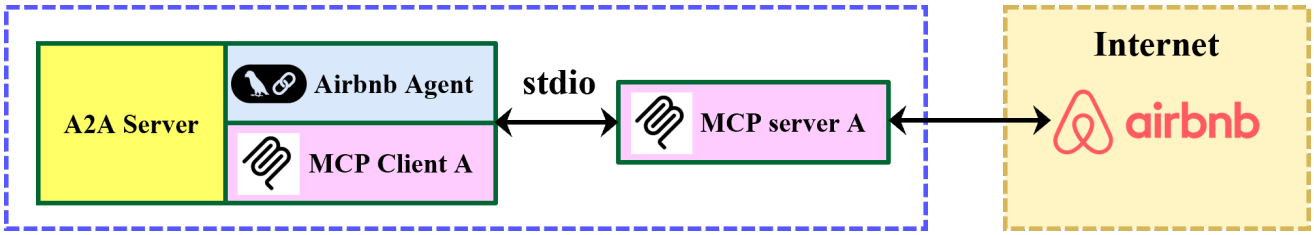
Mỗi Remote Agent được tổ chức như một đơn vị độc lập: có **A2A Server** riêng, tool xử lý riêng, và phiên làm việc ngữ nghĩa (semantic session) riêng. A2A Protocol đóng vai trò **cầu nối** giữa các thực thể độc lập này, đảm bảo sự hợp tác liên-agent mà không cần chia sẻ nội bộ về logic thực thi hay tài nguyên.

Cách tiếp cận này giúp hệ thống có **tính mở rộng cao**: việc bổ sung một agent mới chỉ cần định nghĩa AgentCard và endpoint A2A tương ứng mà không làm ảnh hưởng đến các thành phần khác trong hệ thống.

III.2. Xây dựng các A2A Server

III.2.1. Phần 1: Build Agent Airbnb (airbnb_agent/airbnb_agent.py)

Mục tiêu: Minh họa quá trình tạo ra một agent có khả năng nhận yêu cầu từ người dùng, sử dụng model ngôn ngữ (LLM) + toolset (từ MCP) để phản hồi theo định dạng chuẩn của A2A.



Hình 10: A2A Server - Airbnb Agent

Import các thành phần chính:

```python
from langchain_core.runnables.config import RunnableConfig
from langchain_core.messages import AIMessage, AIMessageChunk
from langgraph.checkpoint.memory import MemorySaver
from langgraph.prebuilt import create_react_agent
from pydantic import BaseModel
from langchain_google_genai import ChatGoogleGenerativeAI
import logging, os, httpx
```

Giải thích các thành phần:

- **langgraph**: Thư viện giúp xây dựng agent có khả năng tương tác theo mô hình *React Agent*, tức là mô hình sử dụng công cụ (tool-augmented reasoning).
- **ChatGoogleGenerativeAI**: Giao diện kết nối đến mô hình ngôn ngữ của Google (Generative AI API) để sinh phản hồi ngôn ngữ tự nhiên.
- **MemorySaver**: Dùng để lưu trạng thái nội bộ (memory) trong quá trình tương tác – đặc biệt hữu ích khi duy trì phiên làm việc theo luồng (session-based) trong hệ thống A2A.

- **RunnableConfig**: Cho phép cấu hình tác vụ theo `thread_id`, giúp agent duy trì đúng ngữ cảnh của hội thoại khi xử lý yêu cầu.
- **AIMessageChunk**: Cho phép phân đoạn kết quả phản hồi, rất hữu ích trong chế độ phản hồi dạng *streaming* (truyền kết quả dần dần về phía người dùng).
- **httpx, logging, os**: Các thư viện bổ trợ thường dùng cho logging, gọi HTTP và xử lý môi trường hệ thống.

Định nghĩa format phản hồi:

```python
class ResponseFormat(BaseModel):
    status: Literal['input_required', 'completed', 'error'] = 'input_required'
    message: str
```

Ý nghĩa trường `status` trong A2A:

- `completed`: Tác vụ đã hoàn tất thành công. A2A sẽ đóng session.
- `input_required`: Agent cần thêm thông tin từ người dùng để tiếp tục.
- `error`: Đã xảy ra lỗi, A2A sẽ hiển thị lỗi cho người dùng.

Lớp `AirbnbAgent` và hàm `__init__`:

```python
class AirbnbAgent:
    SYSTEM_INSTRUCTION = """You are a specialized assistant..."""
    RESPONSE_FORMAT_INSTRUCTION = 'Select status as "completed"...'
    SUPPORTED_CONTENT_TYPES = ['text', 'text/plain']

    def __init__(self, mcp_tools: list[Any]):
        model_name = os.getenv('GOOGLE_GENAI_MODEL')
        self.model = ChatGoogleGenerativeAI(model=model_name)
        self.mcp_tools = mcp_tools
```

Giải thích:

- `SYSTEM_INSTRUCTION`: chỉ dẫn hệ thống để mô hình phản hồi đúng ngữ cảnh – là một yêu cầu khi làm việc với giao thức A2A.
- `RESPONSE_FORMAT_INSTRUCTION`: hướng dẫn định dạng phản hồi – giúp model trả kết quả chuẩn JSON để A2A phân tích.
- `SUPPORTED_CONTENT_TYPES`: xác định các kiểu dữ liệu đầu vào được hỗ trợ.
- Trong `__init__`:

- `mcp_tools`: danh sách các công cụ lấy từ MCP (thường là Tool object).
- `self.model`: khởi tạo mô hình ngôn ngữ Google GenAI.
- Nếu không truyền `mcp_tools`, agent sẽ không thể xử lý tác vụ do không có công cụ đi kèm.

Phương thức `ainvoke()`: khởi chạy Agent và nhận kết quả xử lý không-stream

```python
async def ainvoke(self, query: str, session_id: str) -> dict[str, Any]:
    airbnb_agent_runnable = create_react_agent(
        self.model,
        tools=self.mcp_tools,
        checkpointer=memory,
        prompt=self.SYSTEM_INSTRUCTION,
        response_format=(self.RESPONSE_FORMAT_INSTRUCTION, ResponseFormat),
    )

    config = {'configurable': {'thread_id': session_id}}
    langgraph_input = {'messages': [('user', query)]}

    await airbnb_agent_runnable.ainvoke(langgraph_input, config)

    return self._get_agent_response_from_state(config, airbnb_agent_runnable)
```

Giải thích:

- Tạo Agent theo kiến trúc ReAct sử dụng:
  - `self.model`: mô hình Google GenAI.
  - `self.mcp_tools`: các công cụ lấy từ MCP.
  - `checkpointer=memory`: lưu ngữ cảnh cuộc hội thoại theo session.
  - `response_format`: yêu cầu định dạng đầu ra khớp với `ResponseFormat`.
- `session_id` được truyền vào như `thread_id` – tương ứng với `contextId` trong A2A – giúp giữ ngữ cảnh lâu dài.
- `langgraph_input` chứa thông điệp người dùng gửi.
- Kết quả cuối cùng được truy xuất qua hàm `_get_agent_response_from_state()`.

Hàm `_get_agent_response_from_state()`: Trích xuất kết quả cuối từ Agent

```python
def _get_agent_response_from_state(self, config, agent_runnable) -> dict:
    current_state = agent_runnable.get_state(config)
    structured_response = current_state.values.get('structured_response')
```

Ý nghĩa:

- Hàm này lấy trạng thái hiện tại của Agent thông qua phương thức `get_state()`.
- Trạng thái Agent chứa thông tin đầu ra dưới dạng `structured_response` – đây là kết quả chính thức được A2A sử dụng.
- Nếu không có trường `structured_response`, A2A sẽ fallback sang message cuối cùng (nếu có).

Kết quả trả về chuẩn A2A:

```
{
  "is_task_complete": True/False,
  "require_user_input": True/False,
  "content": "Kết quả trả lời"
}
```

Lưu ý: Đây là định dạng tối ưu để A2A xác định trạng thái task (hoàn tất hay chờ phản hồi tiếp), và hiển thị nội dung cho người dùng.

Hàm `stream()`: Phản hồi theo thời gian thực

```
async def stream(self, query: str, session_id: str) -> AsyncIterable[Any]:
    agent_runnable = create_react_agent(
        self.model,
        tools=self.mcp_tools,
        checkpointer=memory,
        prompt=self.SYSTEM_INSTRUCTION,
        response_format=(self.RESPONSE_FORMAT_INSTRUCTION, ResponseFormat),
    )
    config = {'configurable': {'thread_id': session_id}}
    langgraph_input = {'messages': [('user', query)]}

    async for chunk in agent_runnable.astream_events(langgraph_input, config, version='
                                            v1'):
        event_name = chunk.get('event')
        data = chunk.get('data', {})
        content_to_yield = None

        if event_name == 'on_tool_start':
            content_to_yield = f"Using tool: {data.get('name', 'a tool')}..."
        elif event_name == 'on_chat_model_stream':
            message_chunk = data.get('chunk')
            if isinstance(message_chunk, AIMessageChunk) and message_chunk.content:
                content_to_yield = message_chunk.content

        if content_to_yield:
            yield {
                'is_task_complete': False,
                'require_user_input': False,
                'content': content_to_yield,
            }
```

```
30
31      final_response = self._get_agent_response_from_state(config, agent_runnable)
32      yield final_response
```

Phân tích:

- Hàm `stream` thực hiện tạo một `react agent` từ model và toolset có sẵn.
- Biến `config` định danh phiên làm việc với `thread_id` là `session_id`.
- Biến `langgraph_input` chứa nội dung hội thoại ban đầu từ người dùng.
- Với mỗi sự kiện trong `astream_events`, agent gửi ra từng phần nhỏ (`chunk`):
  - Nếu `event == 'on_tool_start'` → báo tool đang được dùng.
  - Nếu `event == 'on_chat_model_stream'` → lấy nội dung dòng ra từ mô hình.
- Mỗi phần sẽ được yield dưới dạng một dict chuẩn A2A:

```
{
  'is_task_complete': False,
  'require_user_input': False,
  'content': 'Nội dung phản hồi từng phần'
}
```

- Sau khi stream xong, agent lấy trạng thái cuối bằng `_get_agent_response_from_state()` và gửi chunk cuối cùng.

Xử lý lỗi:

- Nếu có lỗi xảy ra trong quá trình stream, trả về phản hồi dạng:

```
{
  'is_task_complete': True,
  'require_user_input': False,
  'content': 'An error occurred during streaming: ...'
}
```

III.2.2. Phần 2: Build AirbnbAgentExecutor (`airbnb_agent/agent_executor.py`)

Mục tiêu: Xây dựng lớp `AirbnbAgentExecutor` – thành phần trung gian giữa A2A Server và Agent. Nó thực hiện các nhiệm vụ chính sau:

- Nhận yêu cầu từ phía A2A Server (thông qua `DefaultRequestHandler`).

- Gọi hàm `agent.stream(...)` để xử lý câu hỏi người dùng.
- Đẩy kết quả phản hồi về `EventQueue` theo định dạng A2A Event, gồm:
  - `TaskStatusUpdateEvent`: Thay đổi trạng thái (working, completed, ...).
  - `TaskArtifactUpdateEvent`: Gửi nội dung phản hồi (artifact) đến client.

Import các thành phần chính:

```python
from a2a.server.agent_execution import AgentExecutor, RequestContext
from a2a.server.events.event_queue import EventQueue
from a2a.types import (
    TaskArtifactUpdateEvent,
    TaskStatusUpdateEvent,
    TaskState,
    TaskStatus,
)
from a2a.utils import (
    new_agent_text_message,
    new_task,
    new_text_artifact,
)
```

Giải thích các thành phần:

- `AgentExecutor`: Lớp nền (base class) cần kế thừa và tuỳ biến để xử lý luồng tác vụ (task).
- `RequestContext`: Đóng gói thông tin từ client gồm `message`, `taskId`, `contextId`.
- `EventQueue`: Hàng đợi nội bộ dùng để gửi các phản hồi (event) từ agent về lại phía A2A Server.
- `TaskStatusUpdateEvent`: Sự kiện cập nhật trạng thái task như `submitted`, `working`, `completed`, `failed`.
- `TaskArtifactUpdateEvent`: Gửi dữ liệu trả lời của agent về client, dưới dạng `artifact`.
- `new_task`, `new_text_artifact`, `new_agent_text_message`: Hàm tiện ích giúp tạo nhanh các đối tượng đúng định dạng cho A2A.

Lớp `AirbnbAgentExecutor`

`AirbnbAgentExecutor` là lớp kế thừa từ `AgentExecutor`, dùng để triển khai cách xử lý yêu cầu người dùng trong hệ thống A2A. Đây là cầu nối giữa A2A server và lớp logic xử lý agent.

Định nghĩa lớp và hàm khởi tạo:

```python
class AirbnbAgentExecutor(AgentExecutor):
    def __init__(self, mcp_tools: list[Any]):
        self.agent = AirbnbAgent(mcp_tools=mcp_tools)
```

Giải thích:

- `AirbnbAgentExecutor`: Subclass của `AgentExecutor`, bắt buộc override lại các hàm như `execute()` hoặc `cancel()` để tùy biến luồng hoạt động.
- `mcp_tools`: Là danh sách công cụ đã được khởi tạo từ các MCP Server. Danh sách này được truyền vào từ phần `main.py`.
- `self.agent`: Khởi tạo một instance của lớp `AirbnbAgent`, lớp này chịu trách nhiệm chính trong việc gọi mô hình LLM và sử dụng công cụ để phản hồi cho người dùng.

Hàm `execute(...)`: Xử lý tác vụ từ phía A2A Server

Đây là phương thức chính để xử lý một task đến từ A2A. Nó nhận đầu vào, gọi agent xử lý bằng `stream()`, và phản hồi kết quả qua `EventQueue`.

Định nghĩa hàm:

```python
@override
async def execute(
    self, context: RequestContext, event_queue: EventQueue
) -> None:
```

Ý nghĩa các tham số:

- `context`: Gói thông tin đầu vào, bao gồm `message`, `taskId`, `contextId`, ...
- `event_queue`: Hàng đợi phản hồi, dùng để gửi các event về A2A server.

Bước 1: Lấy câu hỏi người dùng và task hiện tại

```python
query = context.get_user_input()
task = context.current_task
```

- `query`: Câu hỏi từ người dùng (VD: `"Find Airbnb in Paris"`).
- `task`: Nếu đã tồn tại → tái sử dụng, nếu không → tạo mới ở bước sau.

Bước 2: Tạo mới task nếu chưa có

```python
if not task:
    task = new_task(context.message)
    await event_queue.enqueue_event(task)
```

- `new_task(...)`: Tạo đối tượng task từ message gốc.
- `enqueue_event(task)`: Gửi sự kiện tạo task về server A2A.

Bước 3: Gọi agent xử lý bằng chế độ streaming

```python
async for event in self.agent.stream(query, task.contextId):
```

- Gọi đến hàm `stream(...)` trong `AirbnbAgent`.
- Trả về các dict phản hồi như:
  - `'is_task_complete'`: Task đã hoàn tất chưa?
  - `'require_user_input'`: Có cần người dùng nhập thêm?
  - `'content'`: Nội dung phản hồi.

Bước 4: Gửi phản hồi về A2A theo trạng thái task

- `'is_task_complete'` = `True` → gửi:
  - `TaskArtifactUpdateEvent`
  - `TaskStatusUpdateEvent(state=completed)`
- `'require_user_input'` = `True` → gửi:
  - `TaskStatusUpdateEvent(state=input_required)`
- Ngược lại → phản hồi tạm thời:
  - `TaskStatusUpdateEvent(state=working)`

Ví dụ gửi yêu cầu nhập thêm từ user:

```python
await event_queue.enqueue_event(
    TaskStatusUpdateEvent(
        status=TaskStatus(state=TaskState.input_required),
        message=new_agent_text_message(event['content'], task.agentMessageId)
    )
)
```

Hàm `cancel(...)`: Không hỗ trợ hủy tác vụ giữa chừng

```python
@override
async def cancel(self, context, event_queue):
    raise Exception('cancel not supported')
```

Ý nghĩa:

- Hàm `cancel(...)` được override để xử lý logic khi phía A2A server yêu cầu huỷ một tác vụ đang thực hiện.
- Trong trường hợp này, agent chưa hỗ trợ huỷ giữa chừng nên luôn `raise Exception`.

III.2.3. Phần 3: Build Airbnb A2A Server (airbnb_agent/___main___.py)

Mục tiêu: Khởi tạo và chạy một server tuân theo chuẩn A2A. Server này chịu trách nhiệm:

- Khởi tạo `AgentExecutor` có kết nối tool từ MCP.
- Xây dựng A2A app với lifecycle đầy đủ.
- Đăng ký metadata agent (`AgentCard`) để UI A2A hiểu và tương tác đúng.

Import các thành phần chính:

```python
from a2a.server.apps import A2AStarletteApplication
from a2a.server.request_handlers import DefaultRequestHandler
from a2a.server.tasks import InMemoryTaskStore
from a2a.types import AgentCapabilities, AgentCard, AgentSkill
from agents.airbnb_planner_multiagent.airbnb_agent.agent_executor import
                                            AirbnbAgentExecutor
from agents.airbnb_planner_multiagent.airbnb_agent.airbnb_agent import AirbnbAgent
from langchain_mcp_adapters.client import MultiServerMCPClient
```

Giải thích các thành phần:

- `A2AStarletteApplication`: Tạo ứng dụng ASGI tương thích với A2A Platform.
- `DefaultRequestHandler`: Nhận yêu cầu từ client và gọi đến AgentExecutor để xử lý.
- `AgentCard`, `AgentCapabilities`, `AgentSkill`: Metadata mô tả agent – A2A sẽ sử dụng để hiển thị giao diện.
- `AirbnbAgentExecutor`: Cầu nối giữa A2A server và logic agent.
- `MultiServerMCPClient`: Client dùng để kết nối và tải tools từ các MCP server.

Hàm `app_lifespan(context)`:

```python
@asynccontextmanager
async def app_lifespan(context: dict[str, Any]):
    mcp_client_instance = MultiServerMCPClient(SERVER_CONFIGS)
    mcp_tools = await mcp_client_instance.get_tools()
    context['mcp_tools'] = mcp_tools

    yield  # Cho app tiếp tục chạy

    await mcp_client_instance.__aexit__(None, None, None)
    context.clear()
```

Giải thích:

- Đây là một lifecycle hook.
- MCP client và tools được khởi tạo một lần duy nhất khi app bắt đầu.
- Dọn dẹp khi app tắt (gọi `__aexit__` và `context.clear()`).

Hàm `get_agent_card(...)`:

```python
def get_agent_card(host: str, port: int):
    return AgentCard(
        name='Airbnb Agent',
        description='Helps with searching accommodation',
        url=f'http://{host}:{port}/',
        version='1.0.0',
        defaultInputModes=AirbnbAgent.SUPPORTED_CONTENT_TYPES,
        defaultOutputModes=AirbnbAgent.SUPPORTED_CONTENT_TYPES,
        capabilities=AgentCapabilities(streaming=True, pushNotifications=True),
        skills=[
            AgentSkill(
                id='airbnb_search',
                name='Search Airbnb',
                description='Find places to stay',
                tags=['travel'],
                examples=['airbnb in Paris']
            )
        ]
    )
```

Giải thích:

- Định nghĩa metadata agent để gửi về A2A server.
- Dùng trong hàm khởi tạo `A2AStarletteApplication(...)`.
- Cho phép A2A UI hiển thị agent và tương tác đúng cách.

Hàm `main(...)` – Khởi động A2A Server

Mục tiêu: Hàm `main` thực thi server theo chuẩn A2A, thông qua việc gọi `run_server_async()`.

```
def main(host='localhost', port=10002, log_level='info'):
    asyncio.run(run_server_async(host, port, log_level))
```

Bên trong `run_server_async(...)`:

Gọi `app_lifespan(...)` để khởi tạo tool:

```
async with app_lifespan(app_context):
    mcp_tools = app_context.get('mcp_tools', [])
```

- Load tool từ MCP server một lần.
- Lưu vào biến toàn cục `app_context`.

Khởi tạo `AirbnbAgentExecutor`:

```
airbnb_agent_executor = AirbnbAgentExecutor(mcp_tools=mcp_tools)
```

- Giao tiếp giữa server và agent thông qua hàm `execute()`.

Tạo `RequestHandler`:

```
request_handler = DefaultRequestHandler(
    agent_executor=airbnb_agent_executor,
    task_store=InMemoryTaskStore(),
)
```

- Trung gian giữa client và executor.
- Sử dụng task store nội bộ trong RAM.

Tạo A2A ASGI App:

```
a2a_server = A2AStarletteApplication(
    agent_card=get_agent_card(host, port),
    http_handler=request_handler,
)
asgi_app = a2a_server.build()
```

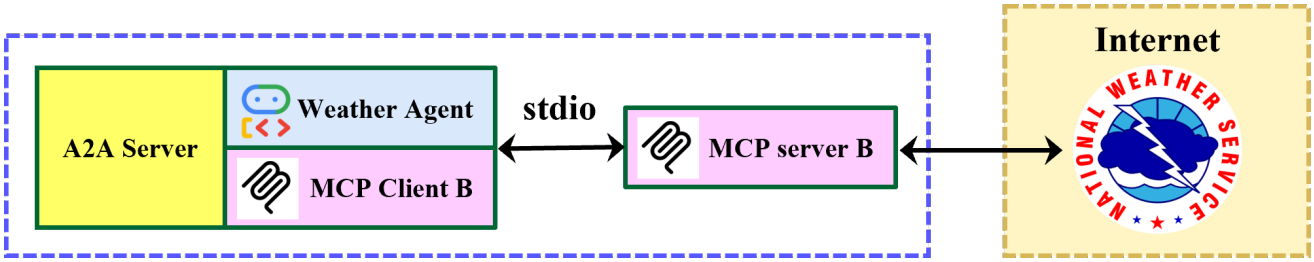
- App này tuân theo chuẩn A2A để tương tác với UI.

Chạy bằng `uvicorn.Server(...)`

```
config = uvicorn.Config(
    app=asgi_app,
    host=host,
    port=port,
    log_level=log_level.lower(),
    lifespan='auto',
)
await uvicorn.Server(config).serve()
```

- Dùng Uvicorn để chạy app ASGI.
- Lifecycle hook (`lifespan='auto'`) cho phép tự động chạy `app_lifespan`.

III.2.4. Phần 4: Build Weather A2A server (weather_agent)



Hình 11: A2A Server - Weather Agent

Tương tự quá trình xây dựng **Airbnb A2A Server**, một A2A server thứ hai được khởi tạo cho **weather_agent**. Đây là một agent dựa trên mô hình ngôn ngữ lớn (LLM-based agent), được phát triển với bộ công cụ **Google ADK SDK**, có khả năng tương tác với người dùng nhằm truy vấn và cung cấp thông tin thời tiết theo thời gian thực. Quá trình khởi tạo tác tử này tuân thủ đầy đủ kiến trúc tích hợp theo chuẩn **Agent-to-Agent (A2A)**.

Các thành phần chính:

- **Lớp khởi tạo**: `LlmAgent` – cung cấp kiến trúc agent tích hợp mô hình ngôn ngữ và công cụ thực thi.
- **Mô hình ngôn ngữ**: `gemini-2.5-flash` – tối ưu hóa cho tốc độ xử lý và chi phí inferencing.
- **Tập công cụ (toolset)**: sử dụng `MCPToolset`, giao tiếp trực tiếp với chương trình `weather_mcp.py` thông qua giao thức `stdio`.
- **Giao tiếp tác vụ**: tác tử sử dụng session để duy trì ngữ cảnh hội thoại, các phản hồi được phát ra theo định dạng chuẩn A2A gồm `TaskStatusUpdateEvent`, `TaskArtifactUpdateEvent`.

Cấu hình giao diện và cổng:

- **Cổng mặc định**: `10001`
- **Chuẩn giao diện**: `A2A-ASGI`, sử dụng lớp `A2AStarletteApplication`
- **Tính năng hỗ trợ**: Streaming nội dung phản hồi, push notification, và khả năng tương thích đầy đủ với nền tảng giao diện người dùng của hệ A2A.

III.3. Xây dựng Host Agent

III.3.1. Phần 1: Build Remote Agent Connection(host_agent/remote_agent_connection.py)

Mục tiêu: Minh họa cách thiết lập kết nối từ xa (*remote connection*) đến các agent con trong kiến trúc Agent-to-Agent (A2A). File này đóng vai trò trung gian giữa host agent và các agent khác (ví dụ: `weather_agent`, `airbnb_agent`) thông qua giao thức A2A tiêu chuẩn.

Import các thành phần chính:

```python
from collections.abc import Callable
import httpx

from a2a.client import A2AClient
from a2a.types import (
    AgentCard,
    SendMessageRequest,
    SendMessageResponse,
    Task,
    TaskArtifactUpdateEvent,
    TaskStatusUpdateEvent,
)
```

Giải thích các thành phần:

- `Callable (collections.abc)`: Kiểu dữ liệu đại diện cho một đối tượng có thể gọi được (callable object), thường dùng để khai báo kiểu hàm hoặc callback.
- `httpx.AsyncClient`: HTTP client bất đồng bộ, dùng để gửi yêu cầu tới các agent từ xa.
- `A2AClient`: Lớp client thuộc SDK A2A, giúp chuẩn hóa giao tiếp với agent qua `AgentCard`.
- `AgentCard`: Thông tin định danh và mô tả của agent từ xa (gồm tên, mô hình, kỹ năng...).
- `SendMessageRequest` / `SendMessageResponse`: Các đối tượng dữ liệu A2A mô tả yêu cầu và phản hồi tác vụ.
- `Task`, `TaskArtifactUpdateEvent`, `TaskStatusUpdateEvent`: Các sự kiện A2A mô tả kết quả hoặc trạng thái của tác vụ.

Định nghĩa các type tiện ích:

```
TaskCallbackArg = Task | TaskStatusUpdateEvent | TaskArtifactUpdateEvent
TaskUpdateCallback = Callable[[TaskCallbackArg, AgentCard], Task]
```

Giải thích:

- `TaskCallbackArg`: Kiểu dữ liệu hợp nhất (Union) đại diện cho một trong ba loại phản hồi được gửi về từ agent con.
- `TaskUpdateCallback`: Một hàm callback được định nghĩa với kiểu đầu vào là `TaskCallbackArg` và `AgentCard`; trả về một `Task`.

Định nghĩa lớp `RemoteAgentConnections`

```
class RemoteAgentConnections:
    """A class to hold the connections to the remote agents."""
```

Lớp bao đóng (*wrapper class*) để:

- Quản lý `A2AClient` đã được khởi tạo.
- Lưu và cung cấp thông tin từ `AgentCard`.
- Giao tiếp với agent con qua phương thức `send_message()`.

Hàm khởi tạo `__init__()`:

```
def __init__(self, agent_card: AgentCard, agent_url: str):
    print(f'agent_card: {agent_card}')
    print(f'agent_url: {agent_url}')
    self._httpx_client = httpx.AsyncClient(timeout=30)
    self.agent_client = A2AClient(
        self._httpx_client, agent_card, url=agent_url
    )
    self.card = agent_card
```

Giải thích:

- Khởi tạo HTTP client bất đồng bộ với timeout 30 giây.
- Tạo `A2AClient` để thực hiện giao tiếp với agent từ xa.
- Lưu lại thông tin `AgentCard` để dùng về sau.

Phương thức `get_agent()`:

```python
def get_agent(self) -> AgentCard:
    return self.card
```

Trả về metadata của agent từ xa – phục vụ mục đích hiển thị hoặc ghi log.

Phương thức chính `send_message()`:

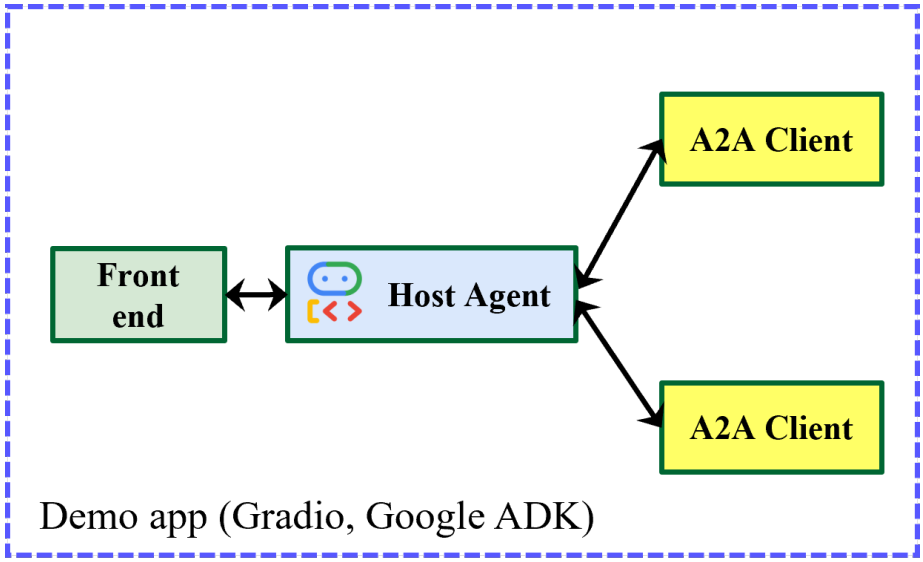
```python
async def send_message(self, message_request: SendMessageRequest) ->
                                                SendMessageResponse:
    return await self.agent_client.send_message(message_request)
```

Đây là phương thức chính để giao tiếp A2A:

- Nhận đầu vào là một `SendMessageRequest` (chuẩn A2A).
- Trả về một `SendMessageResponse` chứa trạng thái và phản hồi từ agent con.

III.3.2. Phần 2: Build RoutingAgent (host_agent/routing_agent.py)

Mục tiêu: Xây dựng tác tử trung tâm (host agent) có khả năng định tuyến yêu cầu của người dùng đến các agent chuyên biệt như `airbnb_agent` hoặc `weather_agent`. Tác tử này sử dụng mô hình ngôn ngữ tự nhiên và công cụ `send_message()` để giao tiếp với các agent con theo kiến trúc Agent-to-Agent (A2A).



Hình 12: Host Agent - Agent chứa các A2A Client .

Import các thành phần chính:

```
import asyncio, json, os, uuid
from typing import Any

import httpx
from a2a.client import A2ACardResolver
from a2a.types import (
    AgentCard,
    SendMessageRequest,
    SendMessageResponse,
    ...
)
from agents.airbnb_planner_multiagent.host_agent.remote_agent_connection import ...
from google.adk import Agent
from google.adk.agents.callback_context import CallbackContext
from google.adk.agents.readonly_context import ReadonlyContext
from google.adk.tools.tool_context import ToolContext
```

Giải thích các thành phần:

- `Agent`: Lớp tác tử chính thuộc SDK ADK của Google – cho phép tạo LLM Agent có công cụ và trạng thái.
- `ToolContext`, `ReadonlyContext`, `CallbackContext`: Giao diện cung cấp quyền truy cập các thành phần khác nhau trong vòng đời agent, bao gồm session, biến môi trường, bộ nhớ, phản hồi...
- `A2ACardResolver`: Công cụ giúp truy xuất `AgentCard` từ các agent con qua HTTP (gồm metadata như tên, mô hình, kỹ năng...).
- `RemoteAgentConnections`: Lớp bao đóng cho phép kết nối, gửi yêu cầu và nhận phản hồi từ các agent con thông qua chuẩn A2A.
- `SendMessageRequest`, `SendMessageResponse`: Định dạng dữ liệu chuẩn A2A để giao tiếp giữa các agent – mô tả yêu cầu và phản hồi tác vụ.

Khởi tạo lớp `RoutingAgent`:

```
class RoutingAgent:
    def __init__(self, task_callback=None):
        self.task_callback = task_callback
        self.remote_agent_connections = {}
        self.cards = {}
        self.agents = ""
```

Lưu trữ danh sách kết nối đến các agent con (`RemoteAgentConnections`), duy trì metadata trong `cards`, và mô tả các agent trong `self.agents`.

Khởi tạo bất đồng bộ các thành phần:

```
async def _async_init_components(self, remote_agent_addresses):
    async with httpx.AsyncClient(timeout=30) as client:
        for address in remote_agent_addresses:
            card = await A2ACardResolver(client, address).get_agent_card()
            remote_connection = RemoteAgentConnections(card, address)
            self.remote_agent_connections[card.name] = remote_connection
            self.cards[card.name] = card
```

Gửi HTTP đến các địa chỉ agent con để lấy thông tin `AgentCard` và thiết lập kết nối.

Tạo agent định tuyến sử dụng SDK:

```
def create_agent(self) -> Agent:
    return Agent(
        model='gemini-2.5-flash',
        name='Routing_agent',
        instruction=self.root_instruction,
        tools=[self.send_message],
        ...
    )
```

Sử dụng mô hình `gemini-2.5-flash`, và khai báo công cụ duy nhất là `send_message()` để tương tác với agent con.

Sinh lệnh hành vi cho agent:

```
def root_instruction(self, context: ReadonlyContext) -> str:
    current_agent = self.check_active_agent(context)
    return f"""
    **Role:** You are a Routing Delegator...
    * **Task Delegation:** Use 'send_message'...
    ...
    * Available Agents: '{self.agents}'
    * Currently Active Seller Agent: '{current_agent['active_agent']}'
    """
```

Tạo hướng dẫn hành vi cho LLM định tuyến, giúp phân phối đúng tác vụ, truyền ngữ cảnh phù hợp và không hỏi lại người dùng nếu không cần thiết.

Kiểm tra agent đang xử lý tác vụ:

```
def check_active_agent(self, context):
    state = context.state
```

```
    if 'session_active' and 'active_agent' in state:
        return {'active_agent': state['active_agent']}
    return {'active_agent': 'None'}
```

Truy xuất agent đang xử lý hiện tại từ `state` của phiên.

Xử lý trước khi gọi model:

```
def before_model_callback(self, callback_context, llm_request):
    state = callback_context.state
    if not state.get('session_active'):
        state['session_id'] = uuid.uuid4()
        state['session_active'] = True
```

Khởi tạo phiên hội thoại nếu chưa tồn tại để giữ ngữ cảnh liên tục.

Liệt kê các agent con khả dụng:

```
def list_remote_agents(self):
    for card in self.cards.values():
        remote_agent_info.append({'name': card.name, 'description': card.description})
```

Chuẩn bị danh sách các agent để chèn vào instruction.

Công cụ chính để giao tiếp – `send_message()`:

```
async def send_message(self, agent_name: str, task: str, tool_context: ToolContext):
    ...
    payload = {
        'message': {
            'role': 'user',
            'parts': [{'type': 'text', 'text': task}],
            'messageId': uuid.uuid4()
        },
        ...
    }
    message_request = SendMessageRequest(...)
    response = await client.send_message(message_request)
    return response.root.result
```

Gửi đoạn hội thoại đến agent con đã chọn thông qua chuẩn A2A, sử dụng lớp `RemoteAgentConnections`.

Hàm tiện trợ để khởi tạo đồng bộ:

```python
def _get_initialized_routing_agent_sync() -> Agent:
    return asyncio.run(_async_main())
```

Cho phép khởi tạo agent một cách đồng bộ trong môi trường không hỗ trợ `async` như Gradio hoặc Streamlit.

III.3.3. Phần 3: Chạy Host Agent (host_agent/___main___.py)

Mục tiêu: Tập tin `__main__.py` đóng vai trò điểm khởi chạy chính cho toàn bộ hệ thống host agent. Tại đây, ta khởi tạo phiên làm việc với `RoutingAgent` và triển khai giao diện `Gradio` để người dùng có thể tương tác trực tiếp với hệ thống thông qua trình duyệt.

Import các thành phần chính:

```python
import asyncio, traceback
from collections.abc import AsyncIterator
from pprint import pformat
import gradio as gr

from agents.airbnb_planner_multiagent.host_agent.routing_agent import root_agent as
                                                routing_agent
from google.adk.events import Event
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types
```

Giải thích các thành phần:

- `routing_agent`: Được khởi tạo sẵn từ file `routing_agent.py`, đây chính là host agent trung tâm.
- `Runner`: Dùng để thực thi agent ADK như một tác tử động trong hệ thống, chịu trách nhiệm xử lý phiên và phản hồi.
- `Event`: Mô hình hoá một sự kiện dữ liệu ADK/A2A – có thể là đoạn phản hồi hoặc tín hiệu kết thúc.
- `InMemorySessionService`: Lưu trạng thái cuộc trò chuyện trong RAM, giúp duy trì ngữ cảnh theo từng phiên làm việc.
- `gradio`: Thư viện Python đơn giản để xây dựng giao diện người dùng tương tác web.

Cấu hình các biến phiên mặc định:

```
APP_NAME = 'routing_app'
USER_ID = 'default_user'
SESSION_ID = 'default_session'
```

Đặt tên ứng dụng, người dùng, và mã phiên mặc định được sử dụng trong hệ thống để khởi tạo tương tác với agent.

Khởi tạo Runner từ Routing Agent:

```
SESSION_SERVICE = InMemorySessionService()
ROUTING_AGENT_RUNNER = Runner(
    agent=routing_agent,
    app_name=APP_NAME,
    session_service=SESSION_SERVICE,
)
```

Ý nghĩa: `Runner` hoạt động như middleware giúp thực thi agent theo từng phiên làm việc, hỗ trợ lưu session qua `InMemorySessionService`.

Xử lý phản hồi từ agent – `get_response_from_agent()`:

```
async def get_response_from_agent(message: str, history: list[gr.ChatMessage]):
    event_iterator = ROUTING_AGENT_RUNNER.run_async(...)

    async for event in event_iterator:
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.function_call:
                    yield gr.ChatMessage(...)
                elif part.function_response:
                    yield gr.ChatMessage(...)
        if event.is_final_response():
            yield gr.ChatMessage(...)
```

Các loại phản hồi được xử lý:

- `function_call`: Agent đang gọi đến một công cụ.
- `function_response`: Phản hồi từ công cụ đã được agent nhận.
- `is_final_response()`: Đánh dấu kết thúc chuỗi phản hồi từ agent.

Nếu có lỗi xảy ra trong quá trình xử lý, hệ thống sẽ ghi lại `traceback` và gửi lỗi về UI cho người dùng biết.

Hàm `main()` – Khởi tạo Gradio app:

```python
async def main():
    await SESSION_SERVICE.create_session(...)

    with gr.Blocks(...) as demo:
        gr.Image(...)
        gr.ChatInterface(
            get_response_from_agent,
            title='A2A Host Agent',
            description='This assistant can help you to check weather and find airbnb accommodation',
        )

    demo.queue().launch(server_name='0.0.0.0', server_port=8083)
```

Ý nghĩa:

- Giao diện Gradio sử dụng `ChatInterface` cho phép trò chuyện hai chiều.
- Hỗ trợ phản hồi dạng streaming nhờ `yield`.
- Giao diện sẽ hiển thị hình ảnh nhận diện, tiêu đề và mô tả rõ ràng.

Khởi chạy ứng dụng:

```python
if __name__ == '__main__':
    asyncio.run(main())
```

Ý nghĩa: Khi chạy trực tiếp tệp, hệ thống sẽ khởi tạo phiên và triển khai Gradio UI tại địa chỉ `http://localhost:8083`.

III.4. Cloning GitHub Repo và chạy Demo

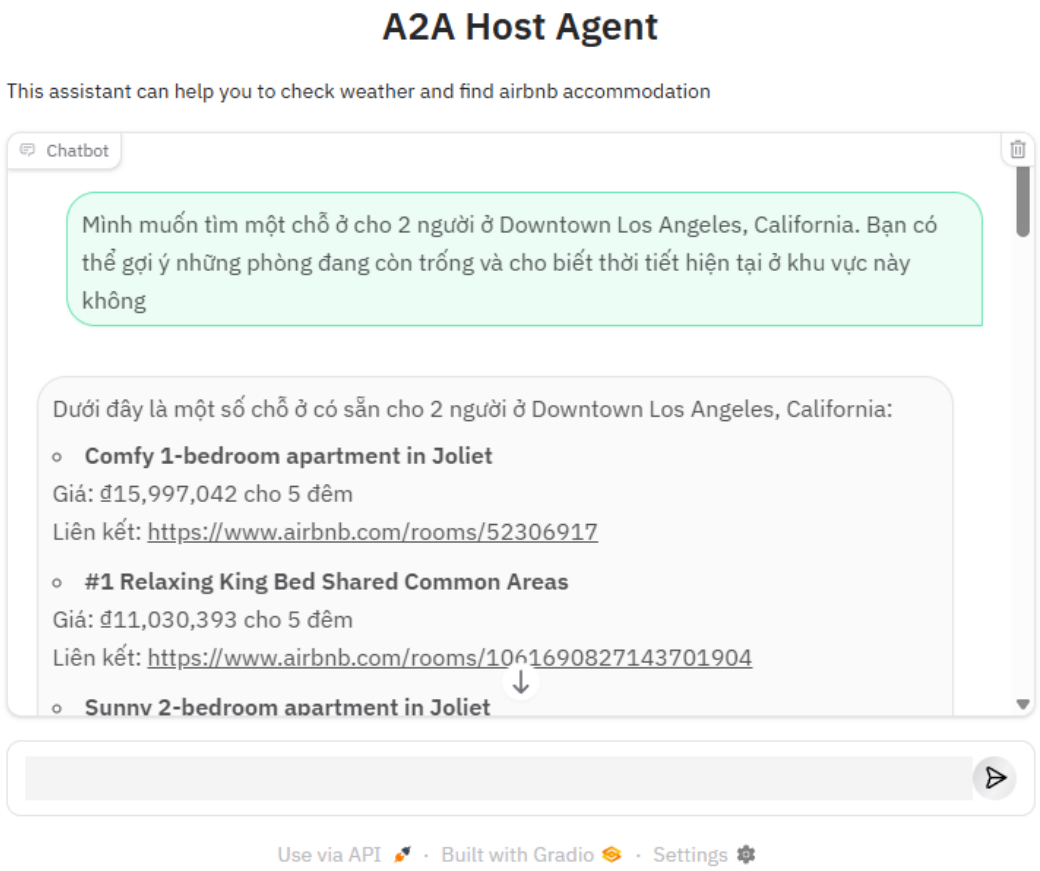
Mã nguồn mẫu được cung cấp tại: `https://github.com/quaghien/A2A-samples`.

Để chạy demo hệ thống multi-agent theo chuẩn A2A, người dùng cần chuẩn bị:

- **Python 3.13**: để chạy `a2a-sdk`.
- **uv**: công cụ quản lý môi trường Python.
- **Node.js (v20)**: phục vụ một số tool (như Airbnb MCP server).
- **.env file**: chứa cấu hình API key và các biến môi trường khác.

Sau đó đọc hướng dẫn chi tiết ở README.md của repo để chạy thử demo.

App UI:



Hình 13: App UI khi chạy demo

IV. Tài liệu tham khảo

[1] A. C. Lab, *Building multi-agent ai app with google's a2a, adk, and mcp*, https://medium.com/ai-cloud-lab/building-multi-agent-ai-app-with-googles-a2a-agent2agent-protocol-adk-and-mcp-a-deep-a94de2237200.

[2] L. Mikami, *Agent2agent and mcp tutorial*, https://huggingface.co/blog/lynn-mikami/agent2agent.

[3] A. Project, *A2a protocol – agent to agent communication standard*, https://a2a-protocol.org/.

[4] A. Project, *A2a protocol sample code*, https://github.com/a2aproject/a2a-samples.

[5] DataCamp, *A2a agent2agent: Standardizing agent communication*, https://www.datacamp.com/blog/a2a-agent2agent.

[6] N. AI, *What is agent2agent (a2a)? a new era of ai agent interaction*, https://blogs.novita.ai/what-is-agent2agent-a2a-a-new-era-of-ai-agent-interaction/.