

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC MỎ - ĐỊA CHẤT



ĐÀO KIM THẮNG

ĐỒ ÁN TỐT NGHIỆP

NGÀNH CÔNG NGHỆ THÔNG TIN
MÃ SỐ: 7480201

TÊN ĐỀ TÀI:
XÂY DỰNG TRỢ LÝ AI CÁ NHÂN – NEURA
(DEVELOPING A PERSONAL AI ASSISTANT - NEURA)

ĐÀO KIM THẮNG

KHOA HỌC MÁY TÍNH ỨNG DỤNG

KHOA 66

Hà Nội - 2025

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC MỎ - ĐỊA CHẤT

ĐỒ ÁN TỐT NGHIỆP

CHUYÊN NGÀNH: KHOA HỌC MÁY TÍNH ỨNG DỤNG

GIÁO VIÊN HƯỚNG DẪN

(Chữ ký)

PGS.TS. LÊ HỒNG ANH

SINH VIÊN THỰC HIỆN

(Chữ ký)

ĐÀO KIM THẮNG

Hà Nội - 2025

LỜI CẢM ƠN

Được sự đồng ý của bộ môn Khoa học máy tính khoa Công nghệ thông tin của trường Đại học Mỏ - Địa chất và giảng viên hướng dẫn khoa học: PGS.TS. Lê Hồng Anh, em đã thực hiện đồ án: “Xây dựng trợ lý AI cá nhân – Neura”.

Để hoàn thành đồ án này, em xin chân thành cảm ơn các thầy cô giảng viên trong khoa Công nghệ thông tin, đặc biệt là bộ môn Khoa học máy tính đã tận tình giảng dạy, hướng dẫn em trong suốt quá trình học tập và nghiên cứu ở trường.

Đặc biệt, xin gửi lời cảm ơn sâu sắc, chân thành tới thầy giáo Lê Hồng Anh, người đã trực tiếp hướng dẫn nghiên cứu khoa học cho em. Trong quá trình thực hiện đồ án, thầy đã tận tình chỉ bảo và truyền đạt những kinh nghiệm và kiến thức khoa học quý báu, đồng thời đưa ra các nhận xét, góp ý giúp em hoàn thành đồ án này một cách khoa học và thực tiễn hơn.

Xin chân thành cảm ơn các anh chị, bạn bè, các thành viên lớp Khoa học máy tính ứng dụng K66_07C đã ủng hộ, giúp đỡ em cả về tinh thần lẫn kiến thức trong suốt quá trình học tập và thực hiện đồ án.

MỤC LỤC

LỜI CẢM ƠN.....	3
MỤC LỤC	4
DANH MỤC TỪ VIẾT TẮT	6
DANH MỤC BẢNG BIÊU.....	7
DANH MỤC HÌNH VẼ	8
CHƯƠNG 1. TỔNG QUAN VỀ ĐỀ TÀI.....	10
1.1. Tính cấp thiết của đề tài.....	10
1.2. Mục tiêu của đề tài.....	11
1.3. Nội dung nghiên cứu.....	11
1.4. Phạm vi nghiên cứu	13
1.5. Bố cục của đồ án.....	13
CHƯƠNG 2. CƠ SỞ LÝ THUYẾT VÀ CÔNG NGHỆ	14
2.1 Các kiến thức nền tảng về Xử lý ngôn ngữ tự nhiên (NLP).....	14
2.1.1 Khái niệm về NLP và ứng dụng.....	14
2.1.2 Các thư viện và công cụ hỗ trợ NLP	16
2.2 Tổng quan về trợ lý ảo	17
2.2.1 Khái niệm và sự phát triển của trợ lý ảo	17
2.2.2 Vai trò, ứng dụng trong đời sống và công việc	18
2.3 Mô hình ngôn ngữ lớn (LLM) và Agent trong Trợ lý ảo	19
2.3.1 Giới thiệu LLM	19
2.3.2. Kiến trúc Agent và hệ thống Multi-Agent trong trợ lý ảo.....	20
2.4 Môi trường phát triển.....	22
2.4.1. Ngôn ngữ lập trình Python	22
2.4.2. Các thư viện hỗ trợ	23
2.4.3. Các dịch vụ và API tích hợp.....	30
2.4.4. Các phần mềm và công cụ.....	31
CHƯƠNG 3. XÂY DỰNG TRỢ LÝ AI CÁ NHÂN.....	34
3.1. Kiến trúc tổng thể trợ lý ảo	34
3.2. Xây dựng Agent Manager.....	35
3.2.1. Thành phần Agent Manager.....	36
3.2.2. Cấu hình Prompt Template và Memory	37

3.3. Xây dựng Agent Calendar	41
3.3.1. Thành phần Agent Calendar.....	42
3.3.2. Cấu hình Prompt Template và Memory.....	58
3.4. Xây dựng Agent Gmail.....	61
3.4.1. Thành phần Agent Gmail.....	62
3.4.2. Cấu hình Prompt Template và Memory.....	66
3.5. Cấu hình LLM và Services.....	69
3.5.1. Cấu hình LLM	69
3.5.2. Cấu hình Google Services	71
3.6. Xây dựng cơ sở dữ liệu.....	72
3.6.1. Lớp ChatMongoDB.....	73
3.6.2. Lớp UserMemoryMongoDB	76
3.7. Phát triển giao diện Web và Bot.....	79
3.7.1. Phát triển giao diện Web	79
3.7.2. Phát triển Telegram Bot.....	84
3.7.3. Phát triển Zalo Bot.....	87
3.8. Testing	89
3.8.1. Tác vụ liên quan đến lịch, sự kiện.....	89
3.8.2. Tác vụ liên quan đến mail	94
3.8.3. Tác vụ khác.....	96
KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN	98
TÀI LIỆU THAM KHẢO	100

DANH MỤC TỪ VIẾT TẮT

Từ viết tắt	Nghĩa tiếng anh	Nghĩa tiếng việt
AI	Artificial Intelligence	Trí tuệ nhân tạo
LLM	Large Language Model	Mô hình ngôn ngữ lớn
RAG	Retrieval-Augmented Generation	Thế hệ tăng cường truy xuất
API	Application Programming Interface	Giao diện lập trình ứng dụng
NLP	Natural Language Processing	Xử lý ngôn ngữ tự nhiên
ML	Machine Learning	Học máy
DL	Deep Learning	Học sâu

DANH MỤC BẢNG BIỂU

DANH MỤC HÌNH VẼ

Hình 2.1. Trợ lý ảo	14
Hình 2.2. Chatbot.....	14
Hình 2.3 Phát hiện ngôn ngữ và dịch thuật	15
Hình 2.4. Minh họa về trợ lý ảo AI trong giao tiếp với người dùng	18
Hình 2.5. Kiến trúc agent	20
Hình 2.6. Các kiến trúc Multi-agent phổ biến	21
Hình 3.1. Sơ đồ kiến trúc tổng thể và luồng hoạt động của trợ lý ảo.....	34
Hình 3.2. Sơ đồ usecase của trợ lý AI	35
Hình 3.3. Biểu đồ hoạt động Agent Manager.....	36
Hình 3.4. Agent Manager	36
Hình 3.5 Memory trong agent manager	37
Hình 3.6. Hàm thực thi agent manager.....	41
Hình 3.7. Biểu đồ hoạt động Agent Calendar	42
Hình 3.8. Agent Calendar	42
Hình 3.9 Hàm tạo sự kiện(create_event_api)	47
Hình 3.11. Hàm lấy danh sách sự kiện(get_multi_calendar_api)	50
Hình 3.10. Hàm lấy thời gian rảnh(get_free_time_api)	50
Hình 3.12. Hàm cập nhật sự kiện (update event)	54
Hình 3.13. Hàm xóa sự kiện(delete_event_api)	57
Hình 3.14. Hàm thực thi agent calendar	61
Hình 3.15. Biểu đồ hoạt động Agent Gmail	62
Hình 3.16. Agent Gmail	62
Hình 3.17 Nhận tham số và lấy danh sách mail từ Gmail API	65
Hình 3.18. Hàm xử lý tóm tắt email (summarize_emails_api)	66
Hình 3.19. Hàm thực thi agent gmail	69
Hình 3.20. Xây dựng lớp LLM.....	70
Hình 3.21. Định nghĩa hàm tóm tắt	70
Hình 3.22. Hàm tạo ReAct Agent	71
Hình 3.23. File Client secrets OAuth 2.0	71
Hình 3.24. Hàm reload token và xây dựng dịch vụ calendar, gmail	72
Hình 3.25. Sơ đồ lớp của hệ thống	72
Hình 3.26. Phương thức save_chat_history trong ChatMongoDB	73
Hình 3.27. Phương thức load_chat_history trong ChatMongoDB	74
Hình 3.28. Phương thức delete_chat_history trong ChatMongoDB	75
Hình 3.29. Phương thức get_all_conversations trong ChatMongoDB	75
Hình 3.30. Phương thức save_info() trong lớp UserMemoryMongoDB	76
Hình 3.31. Ví dụ về thông tin sau khi được lưu vào MongoDB	77
Hình 3.32. Phương thức get_info() trong lớp UserMemoryMongoDB	77
Hình 3.33. Phương thức search_info_by_vector() trong lớp UserMemoryMongoDB.	78
Hình 3.34. Giao diện Web trợ lý AI cá nhân – Neura.....	79

Hình 3.35. Cấu trúc tổng quan của hệ thống web	79
Hình 3.36. Hàm get_conversations() trong route /api/conversations	80
Hình 3.37. Hàm create_conversation() trong route /api/conversations	80
Hình 3.38. Hàm get_conversation() trong route api/conversations/<conversation_id>	81
Hình 3.39. Hàm delete_conversation() trong route /api/conversations/<conversation_id>.....	81
Hình 3.40. Hàm set_active_conversation()	82
Hình 3.41. Hàm add_message() trong route /api/conversations/<conversation_id>/messages.....	83
Hình 3.42. Giao diện hiển thị danh sách hội thoại sử dụng template Jinja2	84
Hình 3.43. Giao diện hiển thị tin nhắn chat sử dụng temolate Jinja2	84
Hình 3.44. Hàm handle_message nhận các tin nhắn từ bot telegram	85
Hình 3.45. Khởi tạo phương thức run_polling() để lắng nghe tin nhắn người dùng	86
Hình 3.46. Hàm reponse_task_schedule gửi thông báo kết quả chạy lịch tự động.....	86
Hình 3.47. Hàm handle_message_zalo nhận các tin nhắn từ bot zalo	87
Hình 3.48. Hàm reponse_task_schedule_zalo gửi thông báo kết quả chạy lịch tự động	88
Hình 3.49. Chuỗi suy luận và hành động của agent manager và calendar.....	89
Hình 3.50. Cuộc hội thoại với chatbot khi tạo sự kiện.....	90
Hình 3.51 .Kết quả chạy Neura khi lấy sự kiện và thời gian rảnh	91
Hình 3.52 Lịch hiển thị đầy đủ được tạo bằng Neura	91
Hình 3.53. Kết quả đổi thời gian sự kiện thành công.....	92
Hình 3.54. Kết quả đổi tiêu đề sự kiện thành công	92
Hình 3.55. Kết quả xóa sự kiện thành công	93
Hình 3.56. Kết quả xóa nhiều sự kiện thành công.....	93
Hình 3.57. Kết quả chạy Neura khi tóm tắt email vào một thời gian cụ thể	94
Hình 3.58. Kết quả chạy Neura khi tóm tắt email vào một khoảng thời gian.....	94
Hình 3.59. Kết quả chạy Neura khi tìm và truy xuất thông tin từ nội dung email.....	95
Hình 3.60. Kết quả chạy Neura khi tìm bản tóm tắt email về một nội dung cụ thể	95
Hình 3.61. Danh sách email từ 6/9 đến 12/9	95
Hình 3.62. Cung cấp thông tin cần ghi nhớ cho chatbot	96
Hình 3.63. Thông tin chatbot đưa ra khi được hỏi đến dữ liệu đã lưu	97
Hình 3.64. Kết quả nhiệm vụ tự động được gửi về bot Telegram và Zalo	97

CHƯƠNG 1. TỔNG QUAN VỀ ĐỀ TÀI

1.1. Tính cấp thiết của đề tài

Trong kỷ nguyên Cách mạng công nghiệp 4.0, trí tuệ nhân tạo (AI) và học máy (Machine Learning) đang thay đổi cách con người làm việc và tương tác với công nghệ. Một trong những ứng dụng tiêu biểu là trợ lý ảo thông minh, hỗ trợ giao tiếp, tra cứu thông tin, quản lý công việc và tự động hóa tác vụ — như Siri, Google Assistant, Alexa hay ChatGPT. Hiện nay, xu hướng AI Agent đang trở thành bước tiến mới của AI, khi không chỉ hiểu và phản hồi hội thoại mà còn có thể lập kế hoạch, ra quyết định và hành động tự động thông qua tích hợp API và công cụ thực tế. Các tập đoàn lớn như OpenAI, Google, Microsoft, Anthropic đều đang tập trung phát triển mạnh mẽ lĩnh vực này. Cụ thể, OpenAI đã ra mắt GPTs – các trợ lý có thể được tùy chỉnh hành vi và khả năng, đồng thời mở rộng với tính năng “Actions” giúp AI tự động thao tác ngoài hội thoại. Google cũng giới thiệu Gemini 2.0 với khả năng truy cập dữ liệu đa phương tiện (multimodal). Microsoft phát triển Copilot, tích hợp trực tiếp vào hệ sinh thái Windows và Office, trong khi Anthropic đưa ra Claude Artifacts, cho phép AI tạo và chỉnh sửa tài liệu, ứng dụng ngay trong hội thoại. Tại Việt Nam, định hướng phát triển AI cũng đang được chú trọng mạnh mẽ. Trong Lễ công bố Chiến lược quốc gia về AI, Bộ trưởng Bộ Thông tin và Truyền thông Nguyễn Mạnh Hùng đã từng phát biểu:

“Tiến tới mỗi người Việt Nam sẽ có một trợ lý của riêng mình, do chính mình nuôi dưỡng, nó chứa toàn bộ hệ tri thức của mình. Khi mình sống thì nó làm trợ lý cho mình, khi mình chết thì nó trở thành chính mình, để con cháu các đời sau vẫn có thể nói chuyện, tâm sự, cho lời khuyên, và chúng ta sẽ trở thành người bất tử.”[1]

Xuất phát từ bối cảnh và xu hướng đó, đề tài “Xây dựng trợ lý AI cá nhân – Neura” được lựa chọn nhằm hướng đến việc phát triển một AI Agent cá nhân hóa thực sự, có khả năng hiểu ngôn ngữ tự nhiên, quản lý lịch trình, đọc và tóm tắt email, ghi nhớ thông tin, và hỗ trợ người dùng trong các hoạt động hàng ngày. Neura không chỉ là chatbot đơn thuần mà được thiết kế để học hỏi, ghi nhớ ngữ cảnh và thực hiện hành động cụ thể thông qua việc tích hợp với các API thực tế như Google Calendar hay Gmail API.

Đề tài mang tính thời sự cao khi phù hợp với xu hướng toàn cầu phát triển AI Agent thông minh, cá nhân hóa và có khả năng hành động, đồng thời đáp ứng định hướng chiến lược về trí tuệ nhân tạo tại Việt Nam. Bên cạnh giá trị ứng dụng thực tiễn, đề tài còn giúp người thực hiện nâng cao kiến thức về NLP, LLM, RAG và kỹ năng triển khai hệ thống thực tế như backend, frontend và tích hợp API. Vì vậy, việc nghiên cứu và phát triển trợ lý AI cá nhân Neura là cần thiết và có ý nghĩa cả về học thuật lẫn thực tiễn, góp phần thúc đẩy xu hướng AI Agent hóa cá nhân trong kỷ nguyên số.

1.2. Mục tiêu của đề tài

Mục tiêu chính của đề tài này là thiết kế, xây dựng và triển khai một trợ lý AI cá nhân, phục vụ nhu cầu quản lý và hỗ trợ công việc của một người dùng. Cụ thể:

- Xây dựng một chatbot AI có khả năng trò chuyện tự nhiên, hỗ trợ trả lời các câu hỏi thường gặp về lịch trình, công việc, hoặc các vấn đề cá nhân khác.
- Quản lý lịch trình (CRUD) bao gồm: tạo lịch hẹn, sửa lịch, xoá lịch, xem lịch trong khoảng thời gian cho trước, xem thời gian rảnh trong khoảng thời gian cho trước thông qua ngôn ngữ tự nhiên .
- Đọc và tóm tắt email, đặc biệt là email có chứa thông tin quan trọng như deadline, lịch hẹn, công việc cần làm. Hỗ trợ gợi ý tạo lịch hẹn ngay từ email nếu phát hiện có mốc thời gian quan trọng.
- Lên lịch các tác vụ tự động và lưu trữ thông tin cá nhân
- Ứng dụng các công nghệ AI mới nhất như Agent, LLM (Large Language Model), RAG (Retrieval-Augmented Generation) để tối ưu hóa khả năng hiểu ngữ cảnh và phản hồi chính xác, đa dạng hơn.
- Triển khai trên nền tảng web và bot, giao diện thân thiện với người dùng, dễ thao tác và có tính mở rộng cao.

1.3. Nội dung nghiên cứu

Để hiện thực hóa mục tiêu trên, đề tài tập trung nghiên cứu và triển khai các nội dung chính như sau:

1. Tổng quan về AI, NLP và trợ lý ảo
 - Nghiên cứu các mô hình ngôn ngữ lớn (LLM) như GPT-4, Gemini hoặc DeepSeek,
 - Nghiên cứu về ReAct Agent, LangChain framework và tools Function Calling cho Agent
 - Khảo sát các nền tảng trợ lý ảo phổ biến và phân tích ưu nhược điểm của từng hệ thống.
2. Xây dựng Agent Calendar
 - Sử dụng LLM cùng với tool Function calling để phát hiện và trích xuất thông tin về lịch trong câu (ví dụ: tiêu đề, thời gian, địa điểm,...).

- Thiết kế CRUD lịch (tạo, đọc, sửa, xoá), lấy thời gian rảnh với Google Calendar API.
- Hỗ trợ các truy vấn tự nhiên: “Lịch tuần này có gì?”, “Ngày mai có lịch gì chưa?”, “Đặt lịch họp vào thứ 6 tuần sau”, “Ngày mai tôi rảnh lúc nào”,...

3. Xây dựng Agent Gmail

- Hỗ trợ các truy vấn bằng ngôn ngữ tự nhiên: “Tóm tắt email từ 10/4 đến 30/4”, “Tóm tắt email hôm qua”,...
- Sử dụng LLM cùng với tool Function calling để phát hiện và trích xuất thông tin về mail (ví dụ: Thời gian, gửi từ ai, ...)
- Tích hợp Gmail API để đọc email và sử dụng LLM để tóm tắt nội dung email, gợi ý tạo lịch dựa trên nội dung email.
- Lưu các email và embedding email để phát triển hệ thống RAG email

4. Xây dựng Agent Manager

- Phân tích ý định người dùng và định tuyến thông minh đến sub-agent chuyên biệt (Gmail hoặc Calendar)
- Trả lời các câu hỏi của người dùng 1 cách thông minh bao gồm các thông tin của người dùng
- Quản lý ngữ cảnh hội thoại và duy trì tính nhất quán trong cuộc trò chuyện, trò chuyện thân thiện.

5. Lưu trữ thông tin cá nhân và truy xuất

- Lưu thông tin cá nhân của người dùng vào cơ sở dữ liệu và agent manager truy vấn mỗi khi cần

6. Thiết kế giao diện web, bot và backend

- Xây dựng giao diện web thân thiện và bot, tối ưu trải nghiệm người dùng, kết nối các module: agent, lịch, email, API backend, đảm bảo tính mở rộng và bảo trì.

7. Đánh giá và thử nghiệm hệ thống

- Đo lường độ chính xác của việc trích xuất thông tin email, tóm tắt email. Đánh giá độ chính xác và tiện ích của module lịch. Đo lường mức độ hài lòng của người dùng qua khảo sát.
- Xây dựng các unittest cho chức năng trích xuất thông tin từ câu truy vấn của người

1.4. Phạm vi nghiên cứu

Đề tài này chỉ tập trung phát triển một trợ lý AI cá nhân (Neura) hỗ trợ một người dùng duy nhất (single-user) trong giai đoạn đầu, hoạt động trên môi trường local (localhost). Các chức năng chính của hệ thống bao gồm quản lý lịch cá nhân (tạo, đọc, sửa, xoá sự kiện và lấy thời gian rảnh dựa trên lịch), đọc, tìm email dựa trên các thông tin (gửi từ ai, nội dung gì,) và tóm tắt email trong một khoảng thời gian nhất định. Hỏi đáp các thông tin cá nhân đã yêu cầu lưu. Trong phạm vi đề tài hệ thống sử dụng các mô hình LLM có sẵn (Gemini) thông qua API thay vì huấn luyện từ đầu vì hạn chế về mặt phần cứng.

1.5. Bộ cục của đồ án

Bộ cục của đồ án được trình bày với các nội dung chính như sau:

Chương 1. Tổng quan về đề tài

Chương này giới thiệu về mục tiêu, ý nghĩa, phạm vi và tóm lược những nội dung của đồ án

Chương 2. Cơ sở lý thuyết và công nghệ

Chương này trình bày các kiến thức nền tảng cần thiết như Agent, NLP, LLM, API tích hợp (Gmail, Calendar), các nghiên cứu liên quan đến trợ lý ảo và các công cụ, thư viện lập trình cho bài toán.

Chương 3. Xây dựng trợ lý AI cá nhân

Chương này trình bày các bước xây dựng các agent (Agent Manager, Agent Gmail, Agent Calendar). Giải thích cách xây dựng luồng, thành phần của các Agent, cấu hình cơ sở dữ liệu và quy trình xây dựng web, telegram, zalo bot và kiểm thử, đánh giá hiệu suất, đo lường độ chính xác và khả năng phản hồi của trợ lý ảo. Đưa ra so sánh kết quả thực tế với mục tiêu đề ra

Kết luận

Phần này tóm tắt lại các kết quả và những đóng góp mà việc thực hiện đề tài này đem lại. Ngoài ra, tổng kết những việc chưa làm được cần khắc phục, đề xuất các phương hướng nghiên cứu tiếp theo, làm cho đề tài trở lên hoàn thiện và hữu ích hơn.

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT VÀ CÔNG NGHỆ

2.1 Các kiến thức nền tảng về Xử lý ngôn ngữ tự nhiên (NLP)

2.1.1 Khái niệm về NLP và ứng dụng

Xử lý ngôn ngữ tự nhiên (tiếng Anh: Natural Language Processing, viết tắt: NLP) đề cập đến lĩnh vực lập trình máy tính để cho phép xử lý và phân tích ngôn ngữ tự nhiên. Từ một thứ cơ bản như một chương trình máy tính để đếm số lượng từ trong một đoạn văn bản, đến thứ phức tạp hơn như một chương trình phục vụ trả lời các câu hỏi của con người hoặc dịch giữa các ngôn ngữ, tất cả đều đủ điều kiện là NLP. Về cơ bản, bất kể mức độ khó, bất kỳ nhiệm vụ nào liên quan đến máy tính xử lý ngôn ngữ thông qua chương trình đủ điều kiện là NLP [2]. NLP chính là chiếc cầu nối giữa ngôn ngữ học truyền thống và công nghệ hiện đại, mở ra khả năng xây dựng các hệ thống thông minh có thể tương tác trực tiếp bằng ngôn ngữ tự nhiên với người dùng, từ đó thu hẹp khoảng cách giữa con người và máy móc.

Trong thực tiễn, NLP ngày nay đã và đang trở thành nền tảng cho nhiều công nghệ tiên tiến, được ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau của đời sống và sản xuất. Một trong những ứng dụng phổ biến và gần gũi nhất chính là chatbot và các trợ lý ảo như Siri của Apple, Alexa của Amazon hay Google Assistant. Những hệ thống này có khả năng tiếp nhận câu hỏi, phân tích ý định của người dùng và đưa ra câu trả lời phù hợp, thậm chí có thể điều khiển các thiết bị thông minh trong gia đình hoặc hỗ trợ đặt lịch, gửi tin nhắn và thực hiện cuộc gọi. Đây là minh chứng rõ nét cho việc NLP đã tiến rất gần đến khả năng giao tiếp tự nhiên như con người.



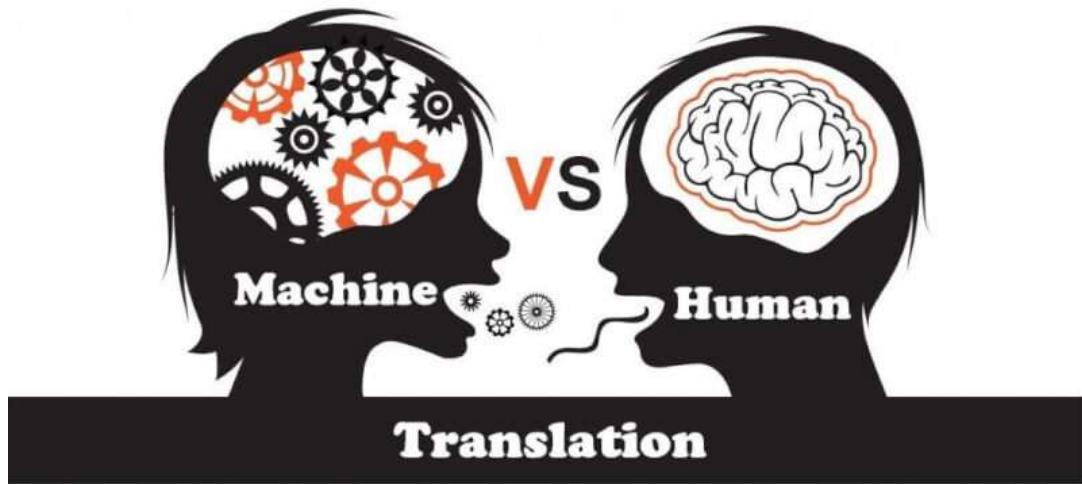
Hình 2.1. Trợ lý ảo



Hình 2.2. Chatbot

Không chỉ dừng lại ở trợ lý ảo, NLP còn được ứng dụng mạnh mẽ trong phân tích dữ liệu xã hội. Các hệ thống phân tích cảm xúc có thể quét qua hàng triệu bài đăng trên mạng xã hội để đánh giá xu hướng dư luận, xác định thái độ tích cực hay tiêu cực của cộng đồng về một sự kiện, sản phẩm hoặc nhân vật. Bên cạnh đó, NLP cũng góp phần quan trọng trong việc phát hiện và lọc bỏ thông tin giả mạo (fake news detection), một vấn đề nóng bỏng trong kỷ nguyên số khi tin giả có thể lan truyền với tốc độ nhanh chóng và gây ra những tác động xã hội nghiêm trọng.

Một mảng ứng dụng nổi bật khác là dịch máy. Các hệ thống dịch tự động như Google Translate hay DeepL đã phát triển vượt bậc trong vài năm gần đây, nhờ kết hợp NLP với các mô hình học sâu (Deep Learning) và mạng nơ-ron dịch máy (Neural Machine Translation). Kết quả là các bản dịch trở nên ngày càng chính xác, tự nhiên và sát nghĩa hơn, thậm chí có khả năng nắm bắt được ngữ cảnh thay vì chỉ dùng lại ở việc dịch từng từ một cách máy móc.



Hình 2.3 Phát hiện ngôn ngữ và dịch thuật

Ngoài ra, NLP còn mang lại nhiều giá trị trong việc tóm tắt văn bản tự động, một công cụ hữu ích cho báo chí, nghiên cứu khoa học và quản lý thông tin, giúp người dùng nhanh chóng nắm bắt được nội dung chính mà không cần đọc toàn bộ tài liệu dài. Các kỹ thuật trích xuất thông tin và thực thể từ văn bản (ví dụ như tên người, địa điểm, ngày tháng, tổ chức) cũng đang được áp dụng rộng rãi trong lĩnh vực tài chính, y tế, pháp lý hay quản trị doanh nghiệp, hỗ trợ quá trình phân tích dữ liệu phi cấu trúc trở nên dễ dàng và hiệu quả hơn.

Bên cạnh đó, NLP còn góp phần nâng cao trải nghiệm người dùng trong các hệ thống nhập liệu và xử lý văn bản thông thường. Những công cụ kiểm tra chính tả, gợi ý từ ngữ hoặc tự động hoàn thành văn bản (autocomplete) đã trở thành tính năng quen thuộc trong các phần mềm soạn thảo và ứng dụng di động, giúp việc nhập liệu trở nên nhanh chóng, chính xác và thuận tiện hơn. Các hệ thống tìm kiếm thông minh dựa trên ngữ nghĩa thay vì chỉ khớp từ khóa cũng là minh chứng rõ nét cho sức mạnh của NLP, khi người dùng có thể tìm được thông tin mong muốn ngay cả khi cách diễn đạt của họ không trùng khớp hoàn toàn với dữ liệu gốc.

Trong lĩnh vực dịch vụ khách hàng, NLP đã mở ra khả năng triển khai các hệ thống chăm sóc khách hàng tự động, có thể tiếp nhận và xử lý yêu cầu 24/7 mà không cần sự can thiệp trực tiếp của nhân viên. Nhờ vậy, doanh nghiệp vừa giảm được chi phí vận hành, vừa nâng cao chất lượng phục vụ, trong khi khách hàng cũng nhận được phản hồi nhanh chóng và nhất quán.

Nhìn chung, NLP không chỉ là một lĩnh vực nghiên cứu học thuật mà đã trở thành công nghệ cốt lõi, đóng vai trò quan trọng trong việc xây dựng các hệ thống thông minh, hỗ trợ con người xử lý lượng dữ liệu khổng lồ, đồng thời cải thiện hiệu quả làm việc và trải nghiệm trong đời sống hàng ngày. Sự phát triển của NLP gắn liền với tiến bộ của trí tuệ nhân tạo nói chung và hứa hẹn tiếp tục mở ra nhiều đột phá mới, từ đó định hình lại cách con người tương tác với máy tính và công nghệ trong tương lai.

2.1.2 Các thư viện và công cụ hỗ trợ NLP

Hiện nay, sự phát triển mạnh mẽ của xử lý ngôn ngữ tự nhiên (NLP) được thúc đẩy đáng kể bởi các thư viện và công cụ mã nguồn mở, vốn giúp cho việc xây dựng, thử nghiệm và triển khai các ứng dụng NLP trở nên dễ dàng và hiệu quả hơn. Các thư viện này không chỉ hỗ trợ những tác vụ cơ bản mà còn cung cấp sẵn các mô hình ngôn ngữ lớn hiện đại, tạo điều kiện thuận lợi cho cả nghiên cứu học thuật lẫn ứng dụng thực tế trong doanh nghiệp.

Một trong những thư viện lâu đời và nổi bật nhất là NLTK (Natural Language Toolkit). Đây được xem là bộ công cụ nền tảng dành cho những người mới bắt đầu nghiên cứu NLP, bởi nó cung cấp hầu hết các chức năng cơ bản như tách từ (tokenization), rút gọn từ (stemming), gán nhãn từ loại (POS tagging), phân tích cú pháp đơn giản, cũng như đi kèm với nhiều tập dữ liệu ngôn ngữ hữu ích. Tuy nhiên, NLTK thường được đánh giá là phù hợp cho mục đích giảng dạy, nghiên cứu học thuật và nguyên mẫu thử nghiệm, hơn là cho các ứng dụng yêu cầu hiệu năng cao trong môi trường sản xuất.

Tiếp đến là spaCy, một thư viện hiện đại được thiết kế với mục tiêu tối ưu về tốc độ và hiệu suất. spaCy hỗ trợ mạnh mẽ các tác vụ như phân tích cú pháp (syntactic parsing), nhận dạng thực thể có tên (Named Entity Recognition – NER), vector hóa văn bản và đặc biệt là tích hợp sẵn nhiều mô hình ngôn ngữ đã được huấn luyện. Nhờ khả năng xử lý dữ liệu văn bản với quy mô lớn và tốc độ nhanh, spaCy được ưa chuộng trong các ứng dụng thực tiễn cần hiệu năng cao, chẳng hạn như hệ thống chatbot thương mại, công cụ phân tích dữ liệu khách hàng, hay dịch vụ xử lý văn bản quy mô lớn.

Trong bối cảnh NLP hiện đại, sự xuất hiện của thư viện Transformers do Hugging Face phát triển đã tạo nên bước ngoặt quan trọng. Đây là thư viện được xây dựng xoay quanh kiến trúc Transformer – nền tảng của hầu hết các mô hình ngôn ngữ lớn (LLMs) hiện nay. Transformers cung cấp sẵn hàng loạt mô hình nổi tiếng như BERT, GPT, T5, LLaMA và nhiều biến thể khác, đồng thời hỗ trợ mạnh mẽ cho quá trình fine-tuning trên dữ liệu chuyên biệt. Ưu điểm lớn của thư viện này là tính linh hoạt, khả năng mở rộng và cộng đồng hỗ trợ rất đông đảo, khiến nó trở thành lựa chọn hàng đầu cho các ứng dụng NLP hiện đại như tóm tắt văn bản, dịch máy, trả lời câu hỏi tự động, hay xây dựng trợ lý ảo thông minh.

Bổ trợ cho thư viện này là Hugging Face Hub, một nền tảng chia sẻ và lưu trữ mô hình AI mã nguồn mở do cộng đồng đóng góp. Tại đây, hàng nghìn mô hình NLP đã được huấn luyện sẵn trên nhiều ngôn ngữ và nhiều tác vụ khác nhau được cung cấp miễn phí, cho phép các nhà phát triển nhanh chóng tích hợp vào hệ thống của mình mà không cần tốn nhiều thời gian huấn luyện lại từ đầu. Hugging Face Hub không chỉ là kho lưu trữ mô hình, mà còn dần trở thành một hệ sinh thái AI mở, hỗ trợ tích hợp dễ dàng vào các pipeline xử lý ngôn ngữ và các ứng dụng thực tiễn như chatbot, trợ lý ảo, hệ thống tìm kiếm nghĩa hay dịch vụ phân tích văn bản trong doanh nghiệp.

Có thể thấy, mỗi thư viện đều mang trong mình những thế mạnh riêng. NLTK thích hợp cho việc học tập và nghiên cứu cơ bản, spaCy phù hợp với các ứng dụng cần tốc độ và hiệu năng, trong khi Transformers cùng Hugging Face Hub mang lại sức mạnh từ các mô hình ngôn ngữ lớn, giúp triển khai nhanh chóng các giải pháp AI tiên tiến. Nhờ sự đa dạng này, các nhà phát triển có thể linh hoạt lựa chọn hoặc kết hợp nhiều công cụ để xây dựng nên những hệ thống NLP tối ưu, đáp ứng tốt nhu cầu của từng bài toán cụ thể.

2.2 Tổng quan về trợ lý ảo

2.2.1 Khái niệm và sự phát triển của trợ lý ảo

Trợ lý ảo (Virtual Assistant) là một hệ thống phần mềm ứng dụng các công nghệ trí tuệ nhân tạo (AI) và xử lý ngôn ngữ tự nhiên (Natural Language Processing – NLP) nhằm hỗ trợ con người thực hiện nhiều tác vụ khác nhau thông qua giao tiếp bằng ngôn ngữ tự nhiên – có thể là giọng nói hoặc văn bản. Với khả năng hiểu, phân tích và phản hồi yêu cầu, trợ lý ảo không chỉ cung cấp câu trả lời trực tiếp mà còn có thể kết nối với nhiều dịch vụ và nền tảng để mang lại trải nghiệm hỗ trợ toàn diện, từ việc tra cứu thông tin, đặt nhắc nhở đến quản lý công việc hằng ngày.

Ngày nay, trợ lý ảo đã vượt xa vai trò của một công cụ trả lời tự động, trở thành người bạn đồng hành thông minh có khả năng giao tiếp, thấu hiểu và hỗ trợ người dùng trong nhiều lĩnh vực của cuộc sống. Nhờ áp dụng các tiến bộ của AI và NLP, hệ thống có thể nhận biết ngữ cảnh, phân tích ý định và phản hồi một cách tự nhiên, gần gũi như khi con người trò chuyện với nhau. Một trong những yếu tố quan trọng giúp trợ lý ảo ngày càng được ưa chuộng là tính cá nhân hóa — khả năng ghi nhớ thói quen, sở thích và lịch sử tương tác để đưa ra phản hồi phù hợp hơn với từng người dùng. Nhờ đó, trải nghiệm sử dụng trở nên thân thiện và hiệu quả hơn, thay vì chỉ dừng lại ở những phản hồi rập khuôn.

Trợ lý ảo hiện diện rộng rãi trên nhiều nền tảng, từ điện thoại thông minh, máy tính cá nhân đến các thiết bị IoT hay hệ thống trong doanh nghiệp. Những sản phẩm như Siri (Apple), Google Assistant, Alexa (Amazon) hay ChatGPT là minh chứng rõ rệt cho sự phát triển nhanh chóng của công nghệ này.

Sự tiến bộ của trợ lý ảo gắn liền với những đột phá trong lĩnh vực trí tuệ nhân tạo, đặc biệt là sự ra đời của các mô hình ngôn ngữ lớn (Large Language Models – LLM). Nếu như các thế hệ trợ lý ảo ban đầu chỉ có thể trả lời những câu hỏi đơn giản và giới hạn trong phạm vi hẹp, thì các hệ thống hiện đại ngày nay đã có thể duy trì hội thoại theo ngữ cảnh, quản lý lịch trình cá nhân, xử lý email, đề xuất nội dung, thậm chí



Hình 2.4. Minh họa về trợ lý ảo AI trong giao tiếp với người dùng

phân tích dữ liệu và hỗ trợ ra quyết định. Nhờ sự kết hợp giữa NLP, học máy (Machine Learning) và các nền tảng dịch vụ thông minh, trợ lý ảo đang dần trở thành một phần thiết yếu trong đời sống cá nhân, công việc và hoạt động doanh nghiệp. Sự phát triển của trợ lý ảo gắn liền với những bước tiến vượt bậc trong lĩnh vực trí tuệ nhân tạo, đặc biệt là sự ra đời và hoàn thiện của các mô hình ngôn ngữ lớn (Large Language Models – LLM). Nếu như những phiên bản sơ khai của trợ lý ảo chỉ có khả năng trả lời các câu hỏi mang tính chất cố định, đơn giản và giới hạn trong một số lĩnh vực nhất định, thì ngày nay, các hệ thống hiện đại đã có thể duy trì hội thoại theo ngữ cảnh, quản lý lịch trình cá nhân, xử lý email, gợi ý nội dung giải trí, thậm chí còn tham gia vào quá trình phân tích dữ liệu và hỗ trợ ra quyết định. Nhờ sự kết hợp giữa NLP, học máy (Machine Learning), và các nền tảng dịch vụ thông minh, trợ lý ảo đang dần trở thành công cụ không thể thiếu trong đời sống cá nhân, công việc và hoạt động của doanh nghiệp.

2.2.2 Vai trò, ứng dụng trong đời sống và công việc

Trợ lý ảo ngày càng được ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau của đời sống hiện đại. Trong sinh hoạt cá nhân, chúng có thể giúp con người quản lý lịch trình một cách khoa học, đặt nhắc nhở để không bỏ lỡ những công việc quan trọng, điều khiển các thiết bị thông minh trong gia đình chỉ bằng giọng nói, đồng thời gợi ý các hoạt động giải trí phù hợp như nghe nhạc, xem phim hoặc đọc sách. Trong môi

trường công việc, trợ lý ảo hỗ trợ sắp xếp lịch họp, quản lý email, ghi chú những nội dung cần thiết và cung cấp khả năng tìm kiếm thông tin nhanh chóng, từ đó giúp nhân viên tiết kiệm thời gian và nâng cao hiệu quả xử lý công việc.

Đối với doanh nghiệp, các ứng dụng của trợ lý ảo còn mở rộng hơn khi được triển khai dưới dạng chatbot để chăm sóc khách hàng 24/7, tự động hóa quy trình bán hàng, phân tích dữ liệu nhằm dự đoán nhu cầu và xu hướng của thị trường. Trong lĩnh vực giáo dục và nghiên cứu, trợ lý ảo lại đóng vai trò là người đồng hành trong học tập, giúp tra cứu thông tin, dịch thuật tài liệu, hỗ trợ phân tích và xử lý dữ liệu nghiên cứu, qua đó tạo điều kiện cho quá trình học tập và nghiên cứu trở nên thuận lợi hơn.

Nhờ những vai trò đa dạng và thiết thực này, trợ lý ảo ngày càng khẳng định được vị thế như một công cụ quan trọng, góp phần tiết kiệm thời gian, tối ưu hóa hiệu suất làm việc và nâng cao trải nghiệm người dùng trong cả đời sống cá nhân lẫn môi trường chuyên nghiệp.

2.3 Mô hình ngôn ngữ lớn (LLM) và Agent trong Trợ lý ảo

2.3.1 Giới thiệu LLM

Mô hình ngôn ngữ lớn, còn gọi là LLM, là các mô hình học sâu rất lớn, được đào tạo trước dựa trên một lượng dữ liệu khổng lồ. Bộ chuyển hóa cơ bản là tập hợp các mạng nơ-ron có một bộ mã hóa và một bộ giải mã với khả năng tự tập trung. Bộ mã hóa và bộ giải mã trích xuất ý nghĩa từ một chuỗi văn bản và hiểu mối quan hệ giữa các từ và cụm từ trong đó [3]. LLM được huấn luyện bao gồm hàng tỷ câu, đoạn văn, tài liệu, và ngữ cảnh khác nhau trên Internet. Mục tiêu của quá trình huấn luyện này là giúp mô hình học được cách hiểu, sinh và thao tác với ngôn ngữ tự nhiên của con người một cách linh hoạt và chính xác.

Cốt lõi của LLM là kiến trúc Transformer, được giới thiệu bởi nhóm nghiên cứu của Google năm 2017. Transformer giúp mô hình có khả năng học được mối quan hệ giữa các từ trong câu thông qua cơ chế self-attention, từ đó hiểu được ngữ cảnh dài và các phụ thuộc ngữ nghĩa phức tạp. Nhờ cơ chế này, LLM có thể không chỉ xử lý văn bản ở mức từ ngữ, mà còn ở mức ý nghĩa và ngữ cảnh của cả đoạn hội thoại.

Các LLM hiện nay như GPT (OpenAI), Gemini (Google), Claude (Anthropic), LLaMA (Meta) đều được huấn luyện trên quy mô dữ liệu lớn và có hàng chục đến hàng trăm tỷ tham số. Những mô hình này không chỉ có khả năng sinh văn bản tự nhiên, mà còn có thể lập luận, tóm tắt, dịch ngôn ngữ, viết mã lập trình, hay thậm chí hiểu và thực hiện các chỉ dẫn phức tạp bằng ngôn ngữ tự nhiên.

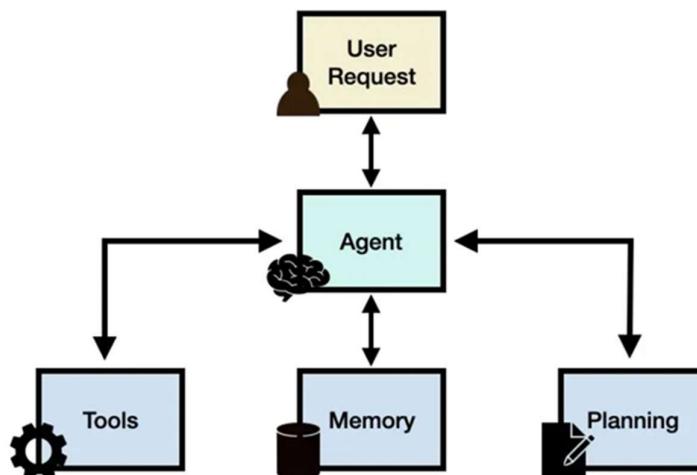
Trong hệ thống trợ lý ảo thông minh, LLM được xem là bộ não trung tâm. Nó chịu trách nhiệm tiếp nhận yêu cầu từ người dùng, phân tích ý định (intent recognition), trích xuất thông tin quan trọng, và đưa ra phản hồi logic, phù hợp ngữ cảnh. Nhờ năng lực hiểu ngữ cảnh đa tầng và khả năng tự điều chỉnh phản hồi dựa trên thông tin trước

đó, LLM có thể mô phỏng được phong cách hội thoại gần với con người, giúp trải nghiệm tương tác trở nên tự nhiên và thân thiện hơn.

Hơn nữa, LLM còn có khả năng kết hợp với cơ sở dữ liệu và các API bên ngoài để thực hiện các hành động cụ thể, ví dụ như tạo lịch, tóm tắt email, tìm kiếm thông tin, hoặc phân loại nội dung. Chính sự linh hoạt và khả năng mở rộng này đã biến LLM trở thành nền tảng quan trọng để xây dựng các ứng dụng thông minh thế hệ mới, đặc biệt là trong các hệ thống trợ lý cá nhân AI, nơi yêu cầu vừa có khả năng hiểu ngôn ngữ tự nhiên, vừa có khả năng hành động thực tế dựa trên ngữ cảnh.

2.3.2. Kiến trúc Agent và hệ thống Multi-Agent trong trợ lý ảo.

Trong hệ thống trợ lý ảo thông minh, kiến trúc Agent được xem như “bộ điều phối trung tâm” giúp kết nối giữa mô hình ngôn ngữ lớn (LLM) và các chức năng cụ thể của hệ thống. Agent có khả năng hiểu sâu ý định của người dùng, tự lập kế hoạch cho các nhiệm vụ phức tạp, quyết định sử dụng các công cụ bên ngoài khi cần thiết, và điều phối toàn bộ luồng xử lý một cách thông minh. Để thực hiện những nhiệm vụ đó Agent thường dựa trên kiến trúc ReAct (Reason + Act). Đây là một phương pháp tiên tiến cho phép hệ thống mô phỏng quá trình tư duy của con người: Lập luận (Reason) để đưa ra quyết định, Hành động (Act) dựa trên quyết định đó, và sau đó Quan sát (Observe) kết quả để tiếp tục chu trình cho đến khi hoàn thành mục tiêu.



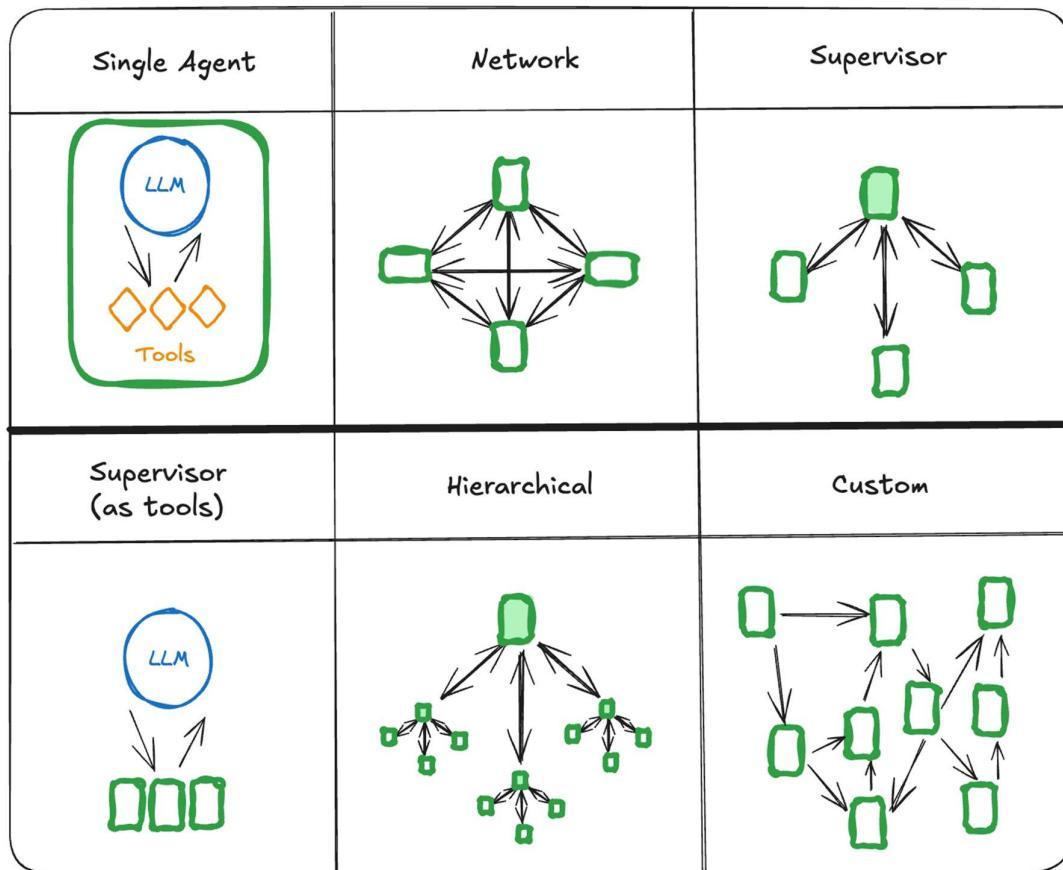
Hình 2.5. Kiến trúc agent

Luồng hoạt động của kiến trúc này là một chu trình lặp đi lặp lại. Khi người dùng đưa ra một yêu cầu, thay vì trả lời ngay lập tức, bộ điều phối Agent sẽ đóng gói yêu cầu đó cùng với danh sách các công cụ có sẵn (như các hàm tương tác với Google Calendar API, Gmail API) và gửi đến mô hình ngôn ngữ lớn (LLM). LLM, đóng vai trò là bộ não lập luận, sẽ phân tích và quyết định xem liệu nó có thể trả lời trực tiếp hay cần sử dụng một công cụ để thu thập thông tin. Nếu cần hành động, LLM sẽ tạo ra một chỉ thị, nêu rõ công cụ cần gọi và các tham số cần thiết. Hệ thống sau đó sẽ thực thi hành động này, ví dụ như gọi đến Google Calendar API để lấy thông tin về lịch trình.

Đồ án tốt nghiệp

Kết quả trả về từ API được xem như một "quan sát" mới. Quan sát này lại được gửi ngược trở lại cho LLM. Dựa trên thông tin mới này, LLM tiếp tục lập luận: hoặc là đã có đủ dữ liệu để đưa ra câu trả lời cuối cùng cho người dùng, hoặc là cần phải thực hiện một hành động khác. Chu trình "Lập luận - Hành động - Quan sát" này tiếp diễn cho đến khi yêu cầu ban đầu của người dùng được giải quyết hoàn toàn.

Trong các hệ thống trợ lý ảo hiện đại, một Agent đơn lẻ thường không thể đáp ứng được toàn bộ các tác vụ phức tạp mà người dùng yêu cầu. Để khắc phục hạn chế đó, kiến trúc đa Agent (Multi-Agent System) đã ra đời. Mô hình này cho phép chia nhỏ hệ thống thành nhiều Agent, mỗi Agent chuyên trách một nhiệm vụ cụ thể như trích xuất thông tin, lập lịch, xử lý email hay phân tích dữ liệu. Nhờ sự chuyên biệt hóa này, từng Agent có thể hiểu sâu hơn về phạm vi nhiệm vụ của mình, từ đó xử lý yêu cầu nhanh chóng, chính xác và hiệu quả hơn. Đồng thời, các Agent có thể phối hợp và trao đổi dữ liệu với nhau, giúp toàn bộ hệ thống vận hành mượt mà, logic và có khả năng mở rộng dễ dàng khi bổ sung thêm tính năng mới.



Hình 2.6. Các kiến trúc Multi-agent phổ biến

Hiện nay, khi xây dựng hệ thống multi-agent thường dựa trên các cấu trúc trên, cụ thể:

- Network (Mạng lưới): mỗi Agent có thể giao tiếp với tất cả các Agent khác. Bất kỳ Agent nào cũng có thể tự quyết định xem nên gọi Agent nào tiếp theo.

- Supervisor (Giám sát): mỗi Agent chỉ giao tiếp với một Agent giám sát trung tâm. Agent giám sát chịu trách nhiệm quyết định xem Agent nào sẽ được gọi tiếp theo.
- Supervisor (tool-calling) – Giám sát (gọi công cụ): đây là một trường hợp đặc biệt của kiến trúc giám sát. Các Agent riêng lẻ được biểu diễn như những “công cụ”. Trong mô hình này, Agent giám sát sử dụng một LLM có khả năng gọi công cụ để quyết định nên gọi Agent nào (tức là công cụ nào) và truyền vào những tham số (arguments) nào.
- Hierarchical (Phân cấp): hệ thống đa Agent có thể được thiết kế với nhiều tầng giám sát — tức là có “giám sát của giám sát”. Đây là phiên bản mở rộng của kiến trúc giám sát, cho phép xây dựng luồng điều khiển phức tạp hơn.
- Custom multi-agent workflow (Luồng làm việc tùy chỉnh): trong mô hình này, mỗi Agent chỉ giao tiếp với một nhóm Agent giới hạn. Một phần của luồng xử lý được xác định sẵn (mang tính cố định), trong khi một số Agent khác có thể tự quyết định xem nên gọi Agent nào tiếp theo.

Trong hệ thống trợ lý AI cá nhân – Neura, kiến trúc được xây dựng dựa trên mô hình Supervisor (tool-calling) – một biến thể phổ biến trong hệ thống đa Agent hiện đại. Trong mô hình này, Agent Manager đóng vai trò là Agent giám sát (Supervisor Agent), chịu trách nhiệm điều phối và quản lý toàn bộ quá trình tương tác giữa người dùng và các Agent chức năng khác. Các Agent chức năng, chẳng hạn như Agent Calendar và Agent Gmail, được biểu diễn dưới dạng các công cụ (tools) mà Agent Manager có thể gọi đến khi cần thực hiện nhiệm vụ cụ thể. Khi người dùng gửi yêu cầu, Agent Manager sẽ sử dụng mô hình ngôn ngữ lớn (LLM) để phân tích ngữ cảnh, xác định loại tác vụ, sau đó lựa chọn công cụ phù hợp và truyền vào các tham số cần thiết (ví dụ: thời gian, chủ đề, người tham gia, nội dung email,...).

Ví dụ, khi người dùng yêu cầu “tạo lịch họp với nhóm vào sáng mai”, Agent Manager sẽ kích hoạt Agent Calendar, cung cấp các tham số được trích xuất từ câu lệnh, và gọi Agent Calendar để xử lý. Tương tự, khi người dùng yêu cầu “tóm tắt các email trong tuần này”, Agent Manager sẽ gọi Agent Gmail để lấy danh sách email tóm tắt, sau đó xử lý và trả kết quả cho người dùng.

2.4. Môi trường phát triển

2.4.1. Ngôn ngữ lập trình Python

Python là một ngôn ngữ lập trình bậc cao, đa năng, được thiết kế để dễ đọc, dễ học và dễ sử dụng. Nó được phát triển bởi Guido van Rossum vào năm 1991 và đã trở thành một trong những ngôn ngữ lập trình phổ biến nhất trên thế giới.

Dưới đây là một số đặc điểm cơ bản về Python:

- Mã nguồn mở: Python là ngôn ngữ mã nguồn mở, nghĩa là nó miễn phí và có thể được sử dụng bởi bất kỳ ai.
- Đễ hiểu: Cú pháp của Python sử dụng khoảng trắng (thụt lè) để định nghĩa các khối mã, không sử dụng dấu ngoặc như nhiều ngôn ngữ lập trình khác.
- Đa năng: Python có thể được sử dụng cho nhiều mục đích khác nhau, bao gồm phát triển ứng dụng web, phân tích dữ liệu, trí tuệ nhân tạo, học máy, tự động hóa công việc, và nhiều ứng dụng khác.
- Thư viện và khung lập trình: Python có một hệ sinh thái mạnh mẽ của thư viện và khung lập trình. Django và Flask cho phát triển web, NumPy và Pandas cho xử lý dữ liệu, TensorFlow và PyTorch cho học máy và trí tuệ nhân tạo, và nhiều thư viện khác.
- Cộng đồng lớn: Python có một cộng đồng rất lớn và đa dạng, với hàng triệu lập trình viên trên khắp thế giới. Nghĩa là có rất nhiều tài liệu, và cộng đồng để hỗ trợ người mới học và những người đã có kinh nghiệm.
- Sự phát triển liên tục: Python liên tục phát triển, với việc ra mắt các phiên bản mới với nâng cấp và cải tiến định kỳ. Python 3.x là phiên bản được khuyến nghị cho mọi dự án mới, trong khi Python 2.x đã bị ngừng hỗ trợ.

2.4.2. Các thư viện hỗ trợ

Thư viện LangChain và các module mở rộng

LangChain là một trong những thư viện mã nguồn mở mạnh mẽ nhất hiện nay hỗ trợ xây dựng các ứng dụng dựa trên mô hình ngôn ngữ lớn (LLM). Mục tiêu của LangChain là cung cấp một khung làm việc (framework) linh hoạt, giúp các nhà phát triển dễ dàng kết hợp LLM với dữ liệu bên ngoài, công cụ, và luồng xử lý phức tạp.

Trong hệ thống trợ lý ảo Neura, LangChain đóng vai trò trung tâm trong việc điều phối các tác vụ như xử lý ngôn ngữ tự nhiên, truy xuất tri thức (RAG), và tương tác với các dịch vụ bên ngoài.

Từ phiên bản 0.1 trở đi, LangChain được tách thành nhiều module độc lập để tăng khả năng mở rộng và tối ưu hiệu suất. Một số module chính được sử dụng trong hệ thống gồm:

- langchain-core: Là thành phần lõi của LangChain, cung cấp các lớp trừu tượng (abstract classes), kiểu dữ liệu, và giao diện chuẩn cho mô hình, chuỗi (Chain), lời nhắc (Prompt), và dữ liệu hội thoại. Đây là nền tảng giúp các module khác tương tác thống nhất và linh hoạt.
- langchain-community: Chứa các tích hợp (integration) được phát triển bởi cộng đồng, bao gồm các kết nối với cơ sở dữ liệu, công cụ tìm kiếm, bộ nhớ, vector store (như Chroma, FAISS), trình tải tài liệu (document loaders) và nhiều mô hình ngôn ngữ khác nhau.

- langchain-openai: Là module chuyên dụng cho việc kết nối với các mô hình của OpenAI, chẳng hạn như GPT-3.5, GPT-4 hoặc GPT-4o. Thư viện này cung cấp các lớp như ChatOpenAI hay OpenAIEmbeddings, giúp dễ dàng tích hợp API OpenAI vào trong chuỗi xử lý của hệ thống.
- langchain (meta package): Là gói tổng hợp để đảm bảo tương thích với các phiên bản cũ. Tuy nhiên, trong các hệ thống mới, việc sử dụng các module riêng biệt như langchain-core, langchain-community và langchain-openai được khuyến khích để đảm bảo hiệu năng và tính bảo trì.

Nhờ sự hỗ trợ của LangChain và các module mở rộng, Neura có thể xây dựng các tác vụ phức tạp dựa trên ngôn ngữ tự nhiên, như phân loại yêu cầu, trích xuất thông tin, truy vấn tri thức, và tự động gọi các API liên quan. Thư viện này giúp rút ngắn đáng kể thời gian phát triển, đồng thời cung cấp cấu trúc chuẩn cho việc xây dựng và mở rộng hệ thống trợ lý ảo trong tương lai.

Thư viện pymongo

PyMongo là thư viện chính thức của MongoDB dành cho ngôn ngữ Python, cho phép các ứng dụng Python kết nối, thao tác và quản lý dữ liệu trong cơ sở dữ liệu MongoDB một cách linh hoạt. Thư viện này cung cấp API trực quan để thực hiện các thao tác như thêm, truy vấn, cập nhật và xóa dữ liệu (CRUD), đồng thời hỗ trợ nhiều tính năng nâng cao như chỉ mục (index), pipeline, và aggregation.

Trong hệ thống trợ lý ảo Neura, PyMongo được sử dụng để lưu trữ và quản lý dữ liệu người dùng, lịch làm việc, lịch sử hội thoại, và các thông tin trích xuất từ email. Việc sử dụng cơ sở dữ liệu MongoDB kết hợp với PyMongo mang lại nhiều ưu điểm nổi bật:

- Lưu trữ linh hoạt: Dữ liệu được lưu dưới dạng tài liệu JSON (document-based), dễ dàng mở rộng và phù hợp với cấu trúc dữ liệu động mà hệ thống trợ lý ảo thường có.
- Tốc độ truy xuất cao: MongoDB và PyMongo được tối ưu cho các thao tác đọc/ghi nhanh, giúp hệ thống phản hồi tốt trong thời gian thực.

Hỗ trợ lưu trữ dữ liệu phi cấu trúc: Cho phép lưu trữ nội dung hội thoại, lịch sử tương tác, và metadata của người dùng mà không cần schema cố định.

Tích hợp dễ dàng với LangChain: Các tác vụ như lưu trữ kết quả tìm kiếm, nội dung tóm tắt hoặc vector embeddings có thể dễ dàng kết nối thông qua PyMongo.

Bộ thư viện Google API

Để tích hợp các dịch vụ của Google Workspace như Google Calendar, Gmail, hay Google Drive vào hệ thống, Neura sử dụng bộ thư viện Google API dành cho Python, bao gồm ba thành phần chính:

google-api-python-client, google-auth-httplib2, và google-auth-oauthlib.

Những thư viện này hỗ trợ hệ thống xác thực người dùng, gửi yêu cầu API, và trao đổi dữ liệu an toàn với các dịch vụ của Google.

- *google-api-python-client* là thư viện trung tâm trong bộ công cụ, cung cấp các lớp và phương thức để tương tác trực tiếp với các API của Google. Thư viện này cho phép xây dựng đối tượng dịch vụ (service) tương ứng với từng API, ví dụ như calendar, gmail, hoặc drive, và thực hiện các thao tác như đọc, ghi, cập nhật hoặc xóa dữ liệu thông qua HTTP request. Trong Neura, *google-api-python-client* được dùng để:

- Lấy danh sách sự kiện, tạo, chỉnh sửa hoặc xóa lịch trên Google Calendar.
- Đọc email thông qua Gmail API.

- *google-auth-httplib2* hỗ trợ quá trình xác thực và ủy quyền giữa ứng dụng Python và máy chủ Google thông qua giao thức HTTP/HTTPS. Nó hoạt động như cầu nối giữa lớp xác thực (*google-auth*) và lớp gửi yêu cầu HTTP (*httplib2*), giúp đảm bảo mọi truy vấn đến API đều được ký và xác minh hợp lệ. Trong hệ thống Neura, *google-auth-httplib2* giúp duy trì phiên đăng nhập an toàn, cho phép trợ lý ảo gửi và nhận dữ liệu từ các API mà không cần xác thực lại liên tục.

- *google-auth-oauthlib* là thư viện hỗ trợ OAuth 2.0 – giao thức tiêu chuẩn dùng để xác thực người dùng trên các dịch vụ Google. Thư viện này cung cấp công cụ tạo luồng xác thực (authentication flow), giúp người dùng đăng nhập bằng tài khoản Google và cấp quyền cho ứng dụng truy cập dữ liệu của họ (ví dụ: quyền xem hoặc chỉnh sửa lịch).

Nhờ sự kết hợp của ba thư viện trên, Neura có thể tích hợp chặt chẽ với hệ sinh thái Google, đóng vai trò cầu nối giữa Neura và các dịch vụ của Google, giúp hệ thống vận hành linh hoạt, an toàn và mang lại trải nghiệm liền mạch khi người dùng tương tác với các công cụ trong Google Workspace.

Thư viện sentence_transformers

Sentence-Transformers là thư viện mã nguồn mở được phát triển dựa trên nền tảng Hugging Face Transformers và PyTorch, chuyên dùng để biểu diễn văn bản (text embedding) dưới dạng vector số có ý nghĩa ngữ nghĩa. Thư viện này cho phép chuyển đổi câu, đoạn văn, hoặc tài liệu thành vector đặc trưng (embedding) mà các mô hình học máy hoặc hệ thống truy xuất tri thức có thể hiểu và so sánh được.

Chức năng chính của Sentence-Transformers:

- Sinh embedding ngữ nghĩa (Semantic Embedding): Mỗi câu được mã hóa thành vector trong không gian nhiều chiều, sao cho những câu có nghĩa tương tự nhau sẽ có khoảng cách nhỏ (theo cosine similarity hoặc Euclidean distance).
- So sánh và truy vấn văn bản: Dễ dàng đo lường mức độ tương đồng giữa hai câu hoặc tìm ra các đoạn văn liên quan nhất trong một tập dữ liệu lớn.
- Hỗ trợ nhiều mô hình mạnh mẽ: Cung cấp hàng trăm mô hình huấn luyện sẵn (pre-trained models) như:
 - all-MiniLM-L6-v2
 - multi-qa-MiniLM-L6-cos-v1
 - halong_embedding

Trong hệ thống Neura, Sentence-Transformers được kết hợp với LangChain và cơ sở dữ liệu vector (MongoDB) sử dụng trong các tác vụ liên quan đến RAG (Retrieval-Augmented Generation), đặc biệt là khi cần:

- Biểu diễn và lưu trữ tri thức người dùng dưới dạng vector embedding.
- Tính toán độ tương đồng ngữ nghĩa giữa câu hỏi hiện tại và các tài liệu đã lưu trữ trong cơ sở tri thức.
- Truy xuất nội dung liên quan để cung cấp cho mô hình ngôn ngữ lớn (LLM) nhằm giảm hiện tượng “ảo giác” (hallucination).

Thư viện APScheduler

APScheduler (viết tắt của Advanced Python Scheduler) là một thư viện Python mạnh mẽ dùng để lên lịch và quản lý các tác vụ tự động (job scheduling). Thư viện này cho phép lập trình viên định nghĩa các tác vụ cần thực thi theo thời gian cụ thể, khoảng thời gian lặp lại, hoặc theo biểu thức cron, tương tự như cron job trong hệ thống Linux.

Các thành phần chính của APScheduler:

- Scheduler (Bộ lập lịch): Thành phần chính chịu trách nhiệm quản lý và thực thi các tác vụ đã được đăng ký. Các loại scheduler phổ biến gồm:
 - BlockingScheduler: Chạy trực tiếp trong tiến trình chính, phù hợp với ứng dụng đơn giản.
 - BackgroundScheduler: Chạy trong nền, thích hợp cho ứng dụng web hoặc hệ thống lớn như Neura.
- Job (Tác vụ): Là hàm Python được định nghĩa để chạy theo lịch, ví dụ gửi email, gọi API, hoặc xử lý dữ liệu.
- Trigger (Bộ kích hoạt): Xác định thời điểm thực thi tác vụ, hỗ trợ nhiều loại:

- date: Chạy một lần vào thời điểm xác định.
- interval: Chạy lặp lại theo khoảng thời gian.
- cron: Lên lịch chi tiết theo giờ, ngày, tháng, tương tự cron của Unix.

Trong hệ thống trợ lý ảo Neura, APScheduler được sử dụng để tự động hóa các tác vụ định kỳ như:

- Gửi thông báo hoặc nhắc nhở cho người dùng vào thời điểm cụ thể.
- Kiểm tra và tóm tắt email hàng ngày.
- Đồng bộ sự kiện giữa Neura và Google Calendar.
- Kích hoạt các quy trình tự động khác trong nền mà không cần sự can thiệp thủ công.

Thư viện Flask

Flask là một micro-framework cho Python, được sử dụng để xây dựng các ứng dụng web. Flask được tạo ra bởi Armin Ronacher vào năm 2010, và mục tiêu của nó là đơn giản hóa việc phát triển web bằng cách cung cấp một bộ công cụ cơ bản và mạnh mẽ để tạo ứng dụng web. Nó rất nhẹ nhàng, dễ sử dụng và linh hoạt, phù hợp cho những dự án nhỏ đến trung bình, và thậm chí là những ứng dụng lớn với khả năng mở rộng.

Dưới đây là một số đặc điểm của Flask:

- Micro-framework: Flask không bao gồm các tính năng như hệ thống quản lý cơ sở dữ liệu hay hệ thống xác thực người dùng mặc định như Django. Thay vào đó, nó cung cấp một cơ sở đơn giản để xây dựng các ứng dụng web và người dùng có thể tự do thêm vào các tính năng mà họ cần.
- Cung cấp công cụ phát triển mạnh mẽ: Flask cung cấp các công cụ phát triển như server phát triển tích hợp, debugger, và hỗ trợ unittesting, giúp việc xây dựng và kiểm thử ứng dụng trở nên dễ dàng.
- Hỗ trợ RESTful API: Flask rất phù hợp cho việc xây dựng các API RESTful nhờ vào tính linh hoạt và cấu trúc đơn giản của nó.
- Mở rộng dễ dàng: Flask hỗ trợ rất tốt việc mở rộng thông qua các extension (mở rộng), ví dụ như hỗ trợ cơ sở dữ liệu với SQLAlchemy, xác thực người dùng với Flask-Login, và nhiều thứ khác.

Thư viện python-telegram-bot

python-telegram-bot là một thư viện mã nguồn mở giúp lập trình viên xây dựng bot Telegram bằng Python một cách đơn giản và hiệu quả. Thư viện này cung cấp giao diện thân thiện với API chính thức của Telegram, cho phép các ứng dụng Python có thể gửi, nhận và xử lý tin nhắn, hình ảnh, tệp tin, hoặc tương tác qua inline button một cách dễ dàng.

Đặc điểm nổi bật của python-telegram-bot:

- Tương thích trực tiếp với Telegram Bot API: Hỗ trợ đầy đủ các phương thức như gửi tin nhắn, hình ảnh, tài liệu, hoặc tạo nút tương tác (InlineKeyboard).
- Cấu trúc handler rõ ràng: Cung cấp các lớp như CommandHandler, MessageHandler, và CallbackQueryHandler giúp dễ dàng quản lý luồng trò chuyện.
- Hỗ trợ bất đồng bộ (Asynchronous): Từ phiên bản 20 trở đi, thư viện được xây dựng dựa trên asyncio, giúp các bot hoạt động mượt mà và phản hồi nhanh hơn.
- Dễ tích hợp với hệ thống khác: Có thể kết hợp cùng Flask, LangChain, hoặc APScheduler để tạo bot AI có khả năng xử lý ngữ cảnh và tự động hóa tác vụ.

Trong hệ thống trợ lý ảo Neura, thư viện python-telegram-bot là cầu nối giữa Neura và người dùng qua nền tảng Telegram, đảm nhiệm các chức năng chính sau:

- Giao tiếp hai chiều: Nhận câu hỏi hoặc yêu cầu của người dùng và gửi phản hồi thông minh từ mô hình AI.
- Thông báo tự động: Gửi nhắc nhở lịch, tóm tắt email, hoặc thông tin quan trọng được lập lịch bởi APScheduler.
- Tích hợp linh hoạt: Làm việc cùng Flask (để xử lý webhook) và LangChain (để gọi mô hình LLM) trong một quy trình thống nhất.

Nhờ tích hợp python-telegram-bot, Neura trở nên linh hoạt hơn, cho phép người dùng tương tác ở bất kỳ đâu, bất kỳ lúc nào, biến Telegram thành một giao diện tương tác di động cho trợ lý AI cá nhân.

Thư viện zalo_bot

zalo-bot là một thư viện mã nguồn mở giúp lập trình viên xây dựng bot Zalo bằng Python một cách nhanh chóng và tiện lợi. Tương tự như python-telegram-bot, thư viện này cung cấp giao diện lập trình thân thiện với Zalo Official Account API, giúp việc kết nối, gửi và nhận tin nhắn trở nên đơn giản hơn rất nhiều.

Đặc điểm nổi bật của zalo-bot:

- Tích hợp trực tiếp với Zalo Official Account API: Hỗ trợ đầy đủ các chức năng gửi – nhận tin nhắn văn bản, hình ảnh, template, và các loại sự kiện tương tác khác từ người dùng Zalo.
- Thiết kế hướng sự kiện (Event-driven): Cung cấp các lớp như MessageHandler, CommandHandler, CallbackHandler, giúp quản lý logic xử lý tin nhắn rõ ràng, tương tự phong cách của python-telegram-bot.
- Hỗ trợ bất đồng bộ (Asyncio): Tương thích với mô hình lập trình bất đồng bộ, giúp bot phản hồi nhanh và xử lý nhiều người dùng cùng lúc.

- Tích hợp linh hoạt: Có thể dễ dàng kết hợp cùng Flask, FastAPI, LangChain hoặc APScheduler để xây dựng hệ thống bot thông minh và tự động hóa.

Trong hệ thống Neura, thư viện zalo-bot là cầu nối giữa Neura và người dùng qua nền tảng Zalo, đảm nhiệm vai trò tương tự Telegram bot nhưng tập trung cho người dùng trong hệ sinh thái Zalo.

Các chức năng chính bao gồm:

- Giao tiếp hai chiều: Nhận tin nhắn hoặc yêu cầu từ người dùng Zalo và gửi phản hồi thông minh từ mô hình AI.
- Thông báo tự động: Gửi nhắc lịch, tóm tắt email, hoặc thông tin quan trọng được thiết lập và quản lý bởi APScheduler.

Nhờ tích hợp zalo-bot, Neura có thể mở rộng khả năng tương tác sang nền tảng Zalo – giúp người dùng Việt Nam dễ dàng tiếp cận và trò chuyện với trợ lý ảo ở mọi nơi, mọi lúc, ngay trong ứng dụng Zalo quen thuộc.

2.4.3. Các dịch vụ và API tích hợp

Hệ thống trợ lý ảo Neura không chỉ hoạt động độc lập mà còn tích hợp nhiều dịch vụ và API bên ngoài nhằm mở rộng khả năng xử lý, giao tiếp và quản lý dữ liệu. Việc sử dụng các API này giúp hệ thống tự động hóa tác vụ, kết nối với nền tảng người dùng quen thuộc và đồng bộ dữ liệu đa chiều. Một số dịch vụ chính được tích hợp gồm:

1. Google Calendar API

Google Calendar API là một trong những dịch vụ cốt lõi của hệ thống Neura, cho phép tương tác trực tiếp với lịch cá nhân của người dùng trên Google. Thông qua API này, Neura có thể:

- Tạo mới, cập nhật, hoặc xóa các sự kiện trên lịch.
- Truy vấn danh sách các sự kiện trong khoảng thời gian nhất định.
- Phát hiện và gợi ý thời gian rảnh cho người dùng.

Tính năng này giúp Neura trở thành trợ lý quản lý thời gian thông minh, có khả năng đồng bộ thông tin lịch làm việc giữa ứng dụng và tài khoản Google của người dùng.

2. Gmail API

Gmail API được tích hợp để đọc, phân loại và tóm tắt nội dung email của người dùng. Hệ thống có thể truy vấn các email trong một khoảng thời gian, nhận dạng các email quan trọng, trích xuất thông tin như thời gian, địa điểm hoặc công việc liên quan, và đề xuất tạo sự kiện lịch tự động dựa trên nội dung thư.

Việc sử dụng Gmail API kết hợp với mô hình ngôn ngữ (LLM) giúp Neura hiểu ngữ cảnh và mục đích thư, từ đó gợi ý hành động phù hợp như “tạo lịch họp”, “gửi phản hồi nhanh”, hay “đánh dấu công việc cần xử lý”.

3. Gemini API

Gemini API là dịch vụ trí tuệ nhân tạo thế hệ mới do Google DeepMind phát triển, thuộc dòng mô hình ngôn ngữ lớn (Large Language Model – LLM), kế thừa và mở rộng khả năng của Bard và PaLM 2.

Gemini không chỉ hỗ trợ xử lý ngôn ngữ tự nhiên mà còn có khả năng đa phương thức (multimodal) – hiểu và sinh nội dung từ nhiều loại dữ liệu như văn bản, hình ảnh, âm thanh, và mã nguồn.

Trong hệ thống Neura, Gemini API đóng vai trò là nền tảng xử lý ngôn ngữ trung tâm, đảm nhiệm các chức năng:

- Phân tích và hiểu truy vấn người dùng: Nhận dạng mục đích câu hỏi, xác định loại tác vụ (tạo lịch, đọc email, hay hỏi đáp thông tin).

- Trích xuất thông tin quan trọng (Entity Extraction): Tách dữ liệu như thời gian, địa điểm, đối tượng, từ ngữ cảnh hội thoại hoặc email để phục vụ quá trình xử lý.
- Sinh phản hồi tự nhiên: Soạn câu trả lời, tóm tắt nội dung hoặc gợi ý hành động tiếp theo cho người dùng.
- Hỗ trợ tích hợp RAG (Retrieval-Augmented Generation): Kết hợp mô hình sinh ngôn ngữ với dữ liệu được truy xuất từ cơ sở tri thức (database hoặc file cá nhân) giúp tăng độ chính xác và giảm hiện tượng “ảo giác” trong phản hồi.

4. MongoDB Atlas (Cloud Database Service)

Bên cạnh các API của Google, Neura còn tích hợp với MongoDB Atlas – dịch vụ cơ sở dữ liệu đám mây của MongoDB. Việc lưu trữ dữ liệu người dùng, lịch làm việc, và lịch sử hội thoại trên nền tảng cloud giúp hệ thống:

- Đảm bảo tính sẵn sàng cao và truy cập từ nhiều thiết bị.
- Dễ dàng mở rộng quy mô dữ liệu.
- Hỗ trợ sao lưu, bảo mật và khôi phục dữ liệu hiệu quả.

MongoDB Atlas được kết nối thông qua thư viện PyMongo, đảm nhiệm vai trò backend lưu trữ dữ liệu động cho toàn bộ hệ thống Neura.

2.4.4. Các phần mềm và công cụ

Phần mềm Visual studio code

Visual Studio Code là trình soạn thảo mã phổ biến được phát triển bởi Microsoft. Nó nhẹ nhàng mạnh mẽ cho cả Windows, macOS và Linux. Nó đi kèm với hỗ trợ tích hợp cho JavaScript, TypeScript và Node.js, đồng thời có kho phần mở rộng phong phú cho các ngôn ngữ khác, chẳng hạn như C++, C#, Java, Python.

Một số tính năng nổi bật của Visual Studio Code:

- Hoạt động như một môi trường phát triển tích hợp: Cung cấp hoàn thành mã thông minh, đề xuất và định dạng mã tự động dựa trên ngữ cảnh cụ thể của ngôn ngữ
- Khả năng tùy chỉnh: Tùy biến cao để phù hợp với sở thích và nhu cầu của từng cá nhân. Nó cho phép người dùng thay đổi chủ đề, tổ hợp phím và cài đặt. Ngoài ra, các nhà phát triển có thể tạo các tiện ích mở rộng hoặc cài đặt các tiện ích mở rộng được phát triển để thêm các chức năng.
- Tích hợp ứng dụng đầu cuối: Tính năng này cho phép chạy các lệnh, xây dựng dự án và thực hiện các tác vụ khác nhau mà không cần chuyển sang một ứng dụng đầu cuối riêng biệt.

- Gỡ lỗi: Cung cấp trải nghiệm gỡ lỗi mạnh mẽ với sự hỗ trợ cho các ngôn ngữ khác nhau. Nó cho phép thiết lập các điểm ngắt, duyệt qua mã, kiểm tra các biến và xử lý các ngoại lệ, giúp xác định và khắc phục sự cố trong mã dễ dàng hơn.
- Kiểm soát phiên bản: Tích hợp git cho phép quản lý mã nguồn trực tiếp. Nó cung cấp các tính năng như quản lý nhánh và giải quyết xung đột, giúp làm việc thuận tiện với các hệ thống kiểm soát phiên bản.
- Chia sẻ trực tiếp: Chia sẻ phiên viết mã với những người khác, cho phép cộng tác theo thời gian thực, đánh giá mã.
- Cộng đồng và tài liệu: Có một cộng đồng lớn và tích cực các nhà phát triển. Tài liệu chính thức cung cấp thông tin chi tiết về các tính năng khác nhau, tùy chọn cấu hình và phát triển tiện ích mở rộng, cùng với các hướng dẫn để giúp người dùng tận dụng tối đa trình chỉnh sửa

Hệ thống Git và GitHub

Git là một hệ thống quản lý phiên bản phân tán (Distributed Version Control System – DVCS), cho phép nhiều lập trình viên cùng làm việc trên cùng một dự án mà vẫn duy trì được tính toàn vẹn của mã nguồn. Mỗi người phát triển đều có một bản sao đầy đủ của kho chứa (repository) trên máy tính của mình, giúp việc theo dõi, sao lưu và khôi phục mã nguồn trở nên nhanh chóng và an toàn.

GitHub là một nền tảng lưu trữ và cộng tác mã nguồn trực tuyến dựa trên Git, cho phép lập trình viên quản lý, chia sẻ và triển khai dự án thông qua kho lưu trữ (repository). GitHub tích hợp các tính năng mở rộng như theo dõi lỗi (Issues), quản lý yêu cầu hợp nhất (Pull Requests), tự động kiểm thử và triển khai (CI/CD), cũng như tài liệu hóa dự án thông qua file README.md.

Một số tính năng nổi bật Git và GitHub

- Quản lý phiên bản (Version Control): Ghi nhận toàn bộ lịch sử thay đổi của mã nguồn thông qua các lần *commit*, cho phép quay lại bất kỳ phiên bản nào khi cần.
- Phân nhánh (Branching): Cho phép tạo các nhánh riêng biệt cho từng tính năng hoặc phần thử nghiệm mà không ảnh hưởng đến nhánh chính (*main*). Điều này giúp phát triển song song và giảm thiểu xung đột mã.
- Hợp nhất mã (Merging): Sau khi hoàn thiện, các nhánh có thể được hợp nhất vào nhánh chính thông qua *pull request*, đảm bảo quá trình kiểm duyệt và kiểm thử trước khi tích hợp.
- Theo dõi lỗi và yêu cầu (Issues & Pull Requests): Hỗ trợ ghi nhận lỗi, đề xuất tính năng mới, và thảo luận trực tiếp giữa các thành viên trong nhóm phát triển.

- Hỗ trợ cộng tác nhóm: Nhiều lập trình viên có thể cùng tham gia vào dự án, chỉnh sửa, nhận xét hoặc đánh giá mã nguồn một cách minh bạch và có kiểm soát.
- Tích hợp CI/CD (Continuous Integration / Continuous Deployment): GitHub cho phép tự động hóa quá trình kiểm thử và triển khai mã nguồn, giúp dự án luôn ở trạng thái hoạt động ổn định.
- Bảo mật và sao lưu dữ liệu: Toàn bộ kho lưu trữ được sao lưu trên nền tảng đám mây, đảm bảo an toàn và dễ dàng truy cập từ mọi thiết bị.

Trong hệ thống trợ lý ảo Neura, Git và GitHub được sử dụng xuyên suốt trong quá trình phát triển để:

- Quản lý mã nguồn và lịch sử thay đổi.
- Hợp tác, mở rộng nhóm phát triển.
- Đảm bảo tính nhất quán giữa các phiên bản.
- Dễ dàng triển khai lại hoặc khôi phục khi cần thiết.

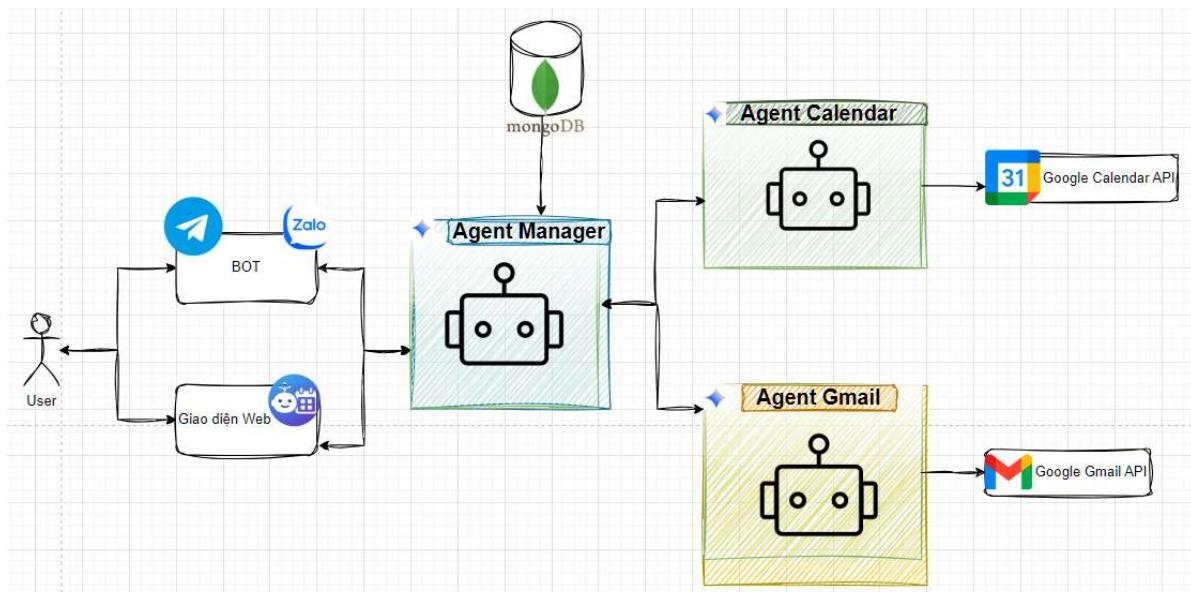
CHƯƠNG 3.

XÂY DỰNG TRỢ LÝ AI CÁ NHÂN

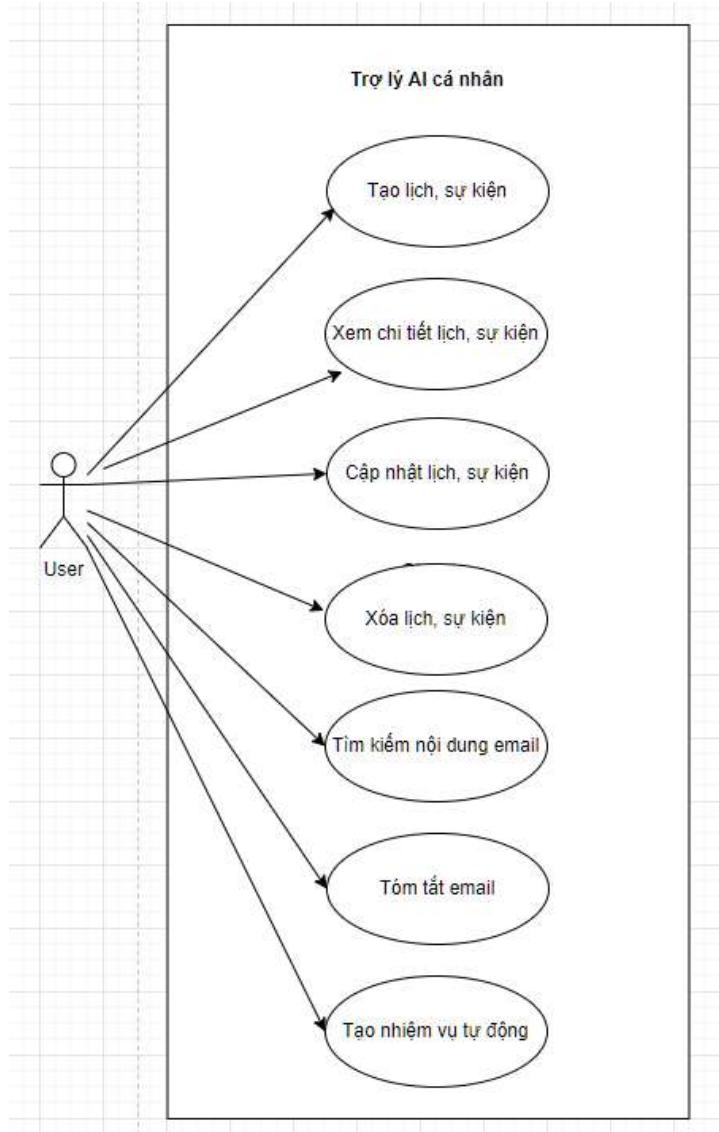
3.1. Kiến trúc tổng thể trợ lý ảo

Hệ thống trợ lý AI cá nhân được thiết kế theo kiến trúc nhiều lớp, trong đó người dùng sẽ trực tiếp tương tác thông qua giao diện chatbot trên nền web hoặc Telegram Bot, Zalo Bot. Tất cả các yêu cầu mà người dùng đưa ra, như tạo, chỉnh sửa, xóa hay kiểm tra lịch, đọc và tóm tắt email, tìm kiếm thông tin hoặc lên lịch tự động, đều được gửi đến Agent Manager. Thành phần này giữ vai trò trung tâm, chịu trách nhiệm phân tích ngữ cảnh, xác định loại yêu cầu và điều phối đến Agent chuyên trách phù hợp.

Các Agent chuyên trách trong hệ thống hiện tại gồm Agent Calendar và Agent Gmail. Agent Calendar đảm nhiệm việc quản lý lịch biểu như tạo sự kiện mới, cập nhật, xóa hoặc truy vấn lịch. Trong khi đó, Agent Gmail hỗ trợ người dùng đọc, tóm tắt, phân loại email. Bên cạnh đó, hệ thống còn sử dụng cơ sở dữ liệu MongoDB để lưu trữ dài hạn, bao gồm lịch sử hội thoại, thông tin đã xử lý từ email và lịch, cũng như các vector embedding phục vụ cho tìm kiếm ngữ nghĩa. Điều này giúp trợ lý có thể gợi nhớ và truy vấn lại thông tin khi cần, tăng tính thông minh và cá nhân hóa.



Hình 3.1. Sơ đồ kiến trúc tổng thể và luồng hoạt động của trợ lý ảo



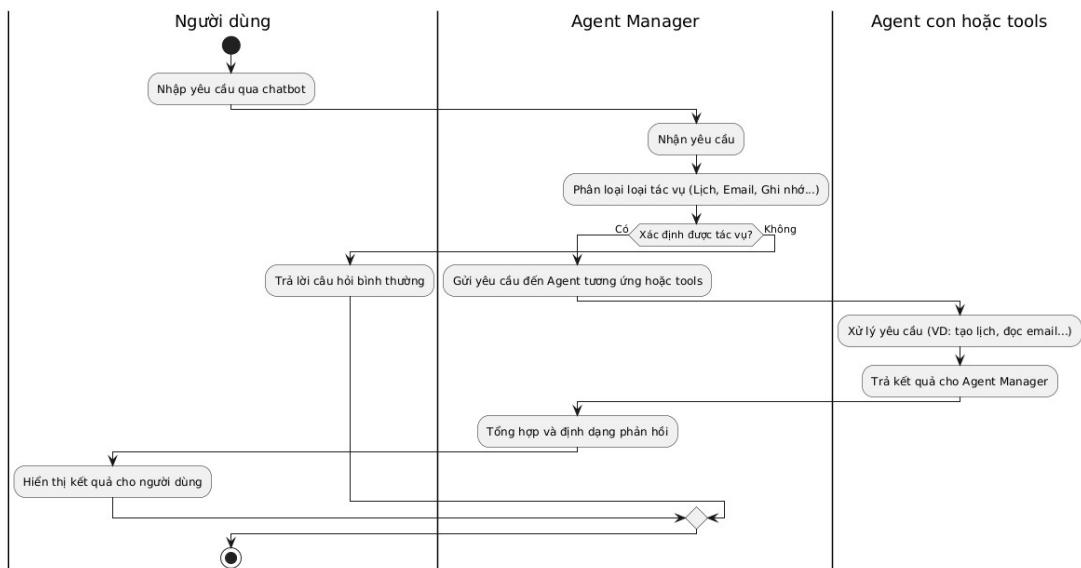
Hình 3.2. Sơ đồ usecase của trợ lý AI

3.2. Xây dựng Agent Manager

Agent Manager giữ vai trò trung tâm trong hệ thống, chịu trách nhiệm tiếp nhận yêu cầu từ người dùng cả từ giao diện Web và bot, phân tích ngữ cảnh và điều phối đến các Agent chuyên trách phù hợp. Chức năng chính của nó là quản lý luồng xử lý, đảm bảo yêu cầu được chuyển đến đúng Agent, đồng thời tổng hợp kết quả phản hồi để gửi lại cho người dùng. Luồng hoạt động của hệ thống diễn ra tuần tự: người dùng đưa ra yêu cầu, Agent Manager phân tích và giao nhiệm vụ, Agent chuyên trách tiến hành xử lý và nếu cần thì gọi đến API bên ngoài hoặc truy vấn cơ sở dữ liệu. Kết quả sau cùng được tổng hợp và phản hồi lại cho người dùng qua chatbot. Cách tổ chức này không chỉ đảm bảo tính rõ ràng, minh bạch trong xử lý mà còn tạo điều kiện thuận lợi

Đồ án tốt nghiệp

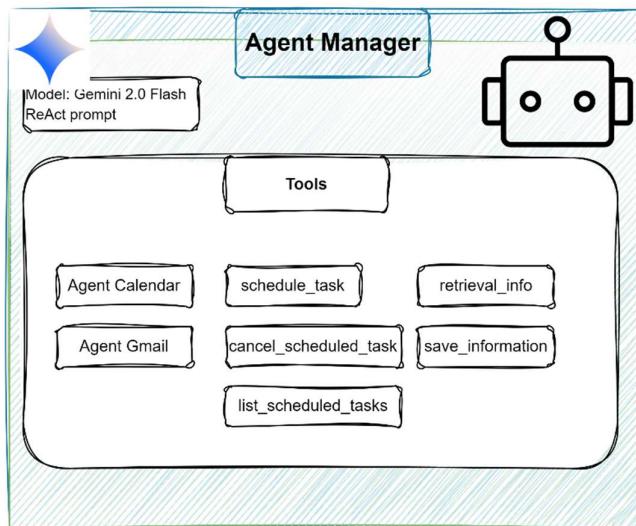
cho việc mở rộng thêm các Agent khác trong tương lai, chẳng hạn như Agent quản lý công việc, Agent ghi chú hay Agent tài chính.



Hình 3.3. Biểu đồ hoạt động Agent Manager

3.2.1. Thành phần Agent Manager

Agent Manager sử dụng LLM Gemini 2.0 Flash với nhiệt độ(temperature) = 0.0 làm bộ não xử lý ngôn ngữ và áp dụng kỹ thuật Prompting ReAct(Reason + Act) cho phép mô hình vừa suy luận để hiểu và phân tích ngữ cảnh, vừa thực hiện hành động phù hợp.



Hình 3.4. Agent Manager

Agent Manager chứa các sub-agent và tools để điều phối để làm các tác vụ khác nhau. Sub-agent như Agent Calendar để giải quyết các vấn đề liên quan đến lịch như tạo, xem, sửa, xóa sự kiện hoặc xem thời gian rảnh của người dùng, Agent Gmail giải quyết các vấn đề đến mail như tóm tắt email, tìm thông tin trong email thông minh,....

Các tools schedule_task để tạo các lịch thực hiện nhiệm vụ tự động vào một thời điểm cụ thể như “Lên lịch tóm tắt email lúc 17h mỗi ngày”, “Lên lịch tạo báo cáo tổng hợp về các cuộc họp hoặc sự kiện trong tuần tới vào 18h Chủ nhật”,....Yêu cầu của người dùng sẽ được đưa tới LLM chuyên trích xuất thời gian, kết quả hàm sẽ nhận được các thông tin như ngày, tháng, giờ, phút, ngày trong tuần và tiến hành lên lịch công việc đó. Tools cancel_scheduled_task để hủy một lịch tự động đã lên như “Hủy lịch tóm tắt email hàng ngày lúc 17h”. Hàm nhận đầu vào là job_id cần hủy, nếu agent không biết job_id nào thì sẽ tiến hành lấy gọi tool list_scheduled_tasks để lấy và tiến hành hủy công việc đó. Tools list_scheduled_tasks để liệt kê toàn bộ danh sách các lịch tự động đã lên trước đó bằng ngôn ngữ tự nhiên để hiển thị cho người dùng.

Ngoài ra agent manager còn có tools retrieve_information để truy vấn thông tin đã lưu về người dùng, thông tin từ các email đã tóm tắt, thông tin có trong các email,... từ cơ sở dữ liệu để cung cấp thêm tri thức cho LLM. Câu truy vấn của người dùng sau khi được truyền vào hàm search_info_by_vector thì sẽ được embedding bằng model halong_embedding thành dạng vector 768 chiều sau đó câu truy vấn sẽ được so sánh với các câu truy vấn bằng thuật toán cosine và lấy ra 5 document có điểm cosine cao nhất cung cấp cho LLM. Chi tiết về hàm search_info_by_vector sẽ được trình bày ở phần cơ sở dữ liệu.

Còn tools save_information để lưu thông tin người dùng vào cơ sở dữ liệu phục vụ cho việc truy vấn thông tin trong tương lai. Tool tạo id bằng thư viện uuid để tạo mã id ngẫu nhiên cho mỗi lần lưu dữ liệu cá nhân.

3.2.2. Cấu hình Prompt Template và Memory

Do Agent Manager trực tiếp tương tác với người dùng nên ta cung cấp bộ nhớ cho agent để có thể nhớ được các câu lệnh rời rạc. Để tránh bị hết bộ nhớ thì giới hạn chỉ lấy 5 tin nhắn gần nhất trong cuộc trò chuyện thông qua phương thức ConversationBufferWindowMemory trong thư viện Langchain.

```
163     memory_manager = ConversationBufferWindowMemory(  
164         memory_key="chat_history",  
165         return_messages=True,  
166         k=5,  # Số lượng tin nhắn trong lịch sử chat  
167     )
```

Hình 3.5 Memory trong agent manager

Đồ án tốt nghiệp

Xây dựng prompt cho Agent để agent có thể hiểu, điều phối, xử lý theo đúng chức năng cung cấp.

Prompt cung cấp cho agent :

“““Bạn là một trợ lý AI thông minh tên Neura, có khả năng phân tích ý định của người dùng và điều phối các agent chuyên biệt.

Nhiệm vụ chính của bạn:

1. Phân tích ý định và nhu cầu của người dùng từ câu hỏi
2. Xác định agent phù hợp nhất để xử lý yêu cầu
3. Chuyển đổi câu hỏi thành định dạng phù hợp cho agent được chọn
4. Điều phối và theo dõi quá trình xử lý:

- *Đưa ra câu trả lời cuối cùng cho người dùng y nguyên agent calendar hoặc agent gmail trả về, không sửa đổi.*

- *Nếu agent calendar hoặc agent gmail cần xác nhận thông tin từ người dùng, hãy hỏi lại người dùng(đưa ra Final Answer y nguyên) để lấy thông tin chính xác hơn không được tự ý quyết định.*

Bạn có quyền truy cập vào các công cụ sau: {tools}

Lịch sử cuộc trò chuyện: {chat_history}

Quy trình xử lý:

- *Nếu liên quan đến lịch (tạo, xem, sửa, xóa sự kiện, tìm thời gian rảnh): sử dụng agent_calendar*

- *Nếu liên quan đến email (tóm tắt, tìm kiếm): sử dụng agent_gmail*

- *Nếu câu hỏi liên quan đến thông tin cá nhân hoặc thông tin về mail thì truy xuất thông tin từ cơ sở kiến thức: sử dụng retrieve_information*

- *Nếu là câu hỏi chung không liên quan đến lịch hoặc email: hãy trả lời bằng ngôn ngữ của người dùng (tiếng Việt hoặc tiếng Anh) một cách tự nhiên và hữu ích*

- *Nếu không rõ ý định, hãy hỏi lại người dùng để lấy thông tin chính xác hơn*

- *Nếu có yêu cầu phức tạp liên quan đến cả hai: chia nhỏ và xử lý tuần tự*

- *Nếu người dùng muốn lên lịch thực hiện một tác vụ tự động, sử dụng 'schedule_task'*

Đò án tốt nghiệp

- Nếu người dùng muốn hủy một lịch tự động, sử dụng 'cancel_scheduled_task' truyền vào job_id, nếu không biết job_id thì sử dụng 'list_scheduled_tasks' để lấy nó

- Nếu người dùng muốn xem danh sách các lịch tự động, sử dụng 'list_scheduled_tasks'

- Nếu câu hỏi mà bạn không xác định được sử dụng agent_calendar hay sử dụng 'schedule_task' thì hãy hỏi lại người dùng để lấy thông tin chính xác hơn.

- Nếu người dùng muốn lưu thông tin thì sử dụng 'save_information'

Hãy sử dụng định dạng sau để trả lời nếu sử dụng tool agent_calendar hoặc agent_gmail:

Question: câu hỏi mà bạn phải trả lời

Thought: phân tích ý định người dùng và xác định agent phù hợp

Action: hành động cần thực hiện, phải là một trong [{tool_names}]

Action Input: câu truy vấn được chuyển đổi cho agent được chọn

Observation: kết quả từ agent giữ nguyên và đưa ra y nguyên không sửa đổi, bao gồm cả các trường rỗng thông tin.

... (quá trình Thought/Action/Action Input/Observation KHÔNG LẮP LAI NHIỀU LẦN, chỉ cần thực hiện một lần duy nhất)

Nếu agent_calendar hoặc agent_gmail cần nhận thông tin từ người dùng, hãy hỏi lại người dùng để lấy thông tin chính xác hơn.

Nếu đã đủ thông tin hoặc gặp lỗi, bạn có thể trả về kết quả/trạng thái cuối cùng:

Final Answer: Đưa ra câu trả lời cuối cùng cho người dùng dựa trên kết quả từ agent

Ví dụ câu hỏi về lịch:

Input: "Tạo lịch họp vào sáng mai lúc 9h"

Thought: Người dùng muốn tạo một sự kiện lịch vào sáng mai lúc 9h.

Tôi sẽ sử dụng agent_calendar để xử lý yêu cầu này.

Action: agent_calendar

Action Input: {{Câu truy vấn của người dùng là: "Tạo lịch họp vào sáng mai lúc 9h"}}

Đồ án tốt nghiệp

Observation: Bạn muốn tạo một sự kiện có tên "Hẹp" vào sáng mai (2025-05-26) từ 09:00 đến 09:00. Địa điểm không được cung cấp. Bạn có muốn tiếp tục không?

Final Answer: Bạn muốn tạo một sự kiện có tên "Hẹp" vào sáng mai (2025-05-26) từ 09:00 đến 09:00. Địa điểm không được cung cấp. Bạn có muốn tiếp tục không?

Kết thúc quá trình đưa Final Answer để hỏi người dùng, nếu người dùng xác nhận thông tin:

Input 2: Đúng rồi

Thought 2: Người dùng đã xác nhận thông tin "Tạo lịch họp vào sáng mai lúc 9h". Tôi sẽ tiến hành tạo sự kiện lịch.

Action 2: agent_calendar

Action Input 2: Người dùng đã xác nhận thông tin "Tạo lịch họp vào sáng mai lúc 9h". Tôi sẽ tiến hành tạo sự kiện lịch.

Observation: Sự kiện lịch đã được tạo thành công vào sáng mai (2025-05-26) từ 09:00 đến 09:00. Địa điểm không được cung cấp

Final Answer 2: Sự kiện lịch đã được tạo thành công vào sáng mai (2025-05-26) từ 09:00 đến 09:00. Địa điểm không được cung cấp

Hãy sử dụng định dạng sau để trả lời các câu hỏi không liên quan đến lịch hoặc email:

Input: câu hỏi của người dùng

Final Answer: câu trả lời tự nhiên và hữu ích cho người dùng

Bắt đầu:

Question: {input}

*{agent_scratchpad}******

Prompt trên sử dụng mẫu prompt ReAct Prompting với cơ chế suy nghĩ rồi đưa ra hành động, quá trình lặp đi lặp lại đến khi agent có đủ thông tin để đưa ra câu trả lời cuối cùng. Prompt nhận 4 biến vào làm giá trị:

- input: Là câu truy vấn của người dùng
- chat_history: Là lịch sử chat giữa người dùng và chatbot
- tools: Là danh sách các tools mà agent có thể truy cập

- agent_scratchpad: Là nơi lưu lại dòng suy nghĩ tạm thời của mô hình trong quá trình suy luận

Từ tất cả thành phần trên(Prompt, Memory, Tools,..) ta xây dựng agent và xây dựng nó thành một hàm thực thi và trả ra kết quả để gọi từ phần khác của chương trình:

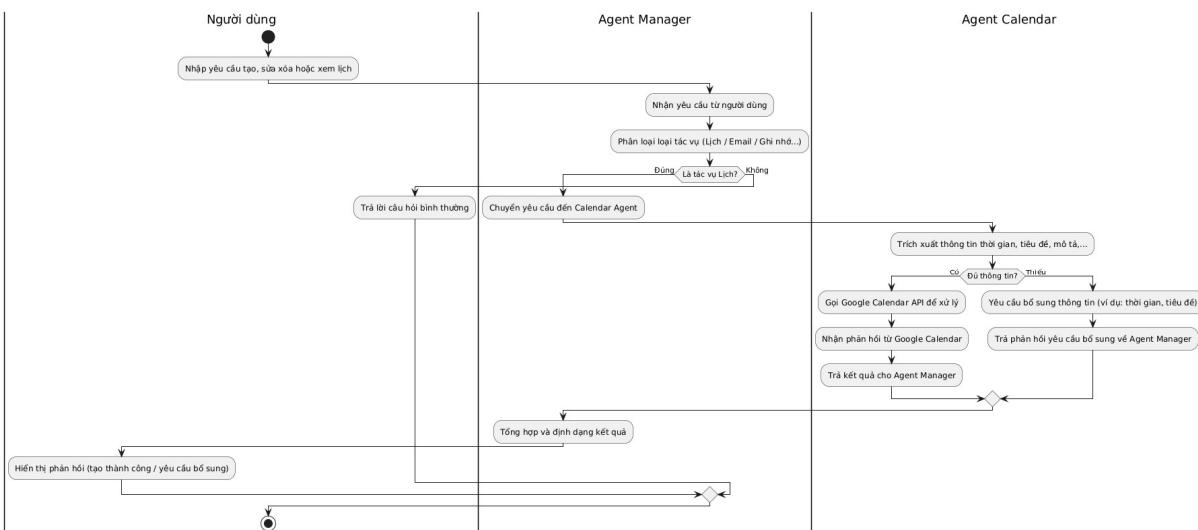
```
150 def agent_manager_executor_func(query, history_chat=None):
151     """
152     Hàm thực thi agent_executor với truy vấn đầu vào và thêm lịch sử chat nếu có.
153     Parameters:
154         query (str): Truy vấn đầu vào từ người dùng.
155         history_chat (list): Lịch sử trò chuyện dạng list (dicts hoặc messages).
156     Returns:
157         str: Kết quả từ agent_executor hoặc thông báo lỗi.
158     """
159     # Đang load lịch sử chat:
160     memory_manager = ConversationBufferWindowMemory(
161         memory_key="chat_history",
162         return_messages=True,
163         k=5,) # Số lượng tin nhắn trong lịch sử chat
164     if history_chat:
165         # Convert lịch sử về messages
166         messages = convert_list_to_messages(history_chat)
167         # Nạp lại vào memory
168         memory_manager.chat_memory.messages = messages
169     else:
170         print("Không có lịch sử chat để nạp vào memory.")
171     # print("\nLịch sử chat đã được nạp vào memory:", memory_manager.chat_memory.messages)
172     # Tạo agent_executor với memory đã cấu hình
173     agent_executor = create_react_agent_executor( tools=tools, memory=memory_manager,
174                                                 prompt_template=prompt_template, option_api=2)
175     print("\nAgent Manager đang xử lý...")
176     result = agent_executor.invoke({"input": query})
177     return result.get("output", "Lỗi trong quá trình thực thi!")
```

Hình 3.6. Hàm thực thi agent manager

3.3. Xây dựng Agent Calendar

Agent Calendar có chức năng xử lý các tác vụ liên quan đến lịch biểu, bao gồm tạo, sửa, xóa và truy vấn sự kiện, thời gian rảnh. Mỗi yêu cầu của người dùng được phân tích bằng LLM để trích xuất thông tin cần thiết, sau đó kết hợp với Google Calendar API để thực hiện thao tác tương ứng. Vai trò của Agent Calendar là đảm bảo quản lý lịch diễn ra tự động, chính xác và đồng bộ, giúp người dùng tối ưu hóa thời gian và duy trì sự gọn gàng trong công việc cũng như sinh hoạt hàng ngày.

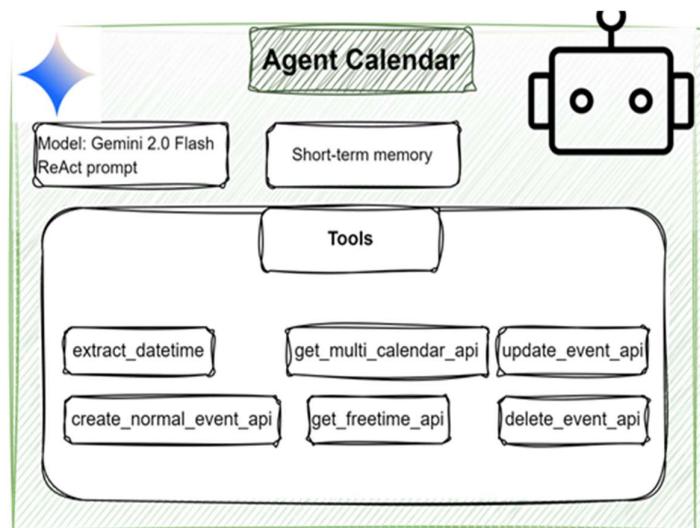
Đồ án tốt nghiệp



Hình 3.7. Biểu đồ hoạt động Agent Calendar

3.3.1. Thành phần Agent Calendar

Agent Calendar sử dụng LLM Gemini 2.0 Flash làm bộ não xử lý ngôn ngữ và áp dụng kỹ thuật Prompting ReAct(Reason + Act) cho phép mô hình vừa suy luận để hiểu và phân tích ngữ cảnh, vừa thực hiện hành động phù hợp cho các yêu cầu liên quan đến lịch, sự kiện. Được cấu hình với tham số nhiệt độ(temperature) = 0.0 giúp LLM thực hiện đúng chức năng, không bị thông tin tránh được tình trạng ảo giác(hallucination) là một trong những tình trạng phổ biến trong LLM. Agent Calendar chứa các tools là extract_datetime, create_normal_event, get_freetime, get_multi_calendar, update_event, delete_event để điều phối để làm các tác vụ khác nhau liên quan đến lịch, sự kiện.



Hình 3.8. Agent Calendar

3.3.1.1. Create_normal_event

Create_normal_event sử dụng model LLM là Gemini-Flash-2.0 với các tham số temperature=0.1 và đầu ra của function calling bao gồm: tiêu đề của sự kiện(title), datetime_ranges chứa danh sách thời gian bắt đầu và kết thúc của các sự kiện(start_datetime, end_datetime) và luật lặp lại chuẩn định dạng iCalendar(rrules), địa điểm của sự kiện(location), thời gian của người dùng có hợp lệ hay không(incorrect_datetime), các tham số về nhắc nhở sự kiện(reminders).

Prompt của create_normal_event để hướng dẫn LLM extract đúng theo các trường:

“Extract information to create an event from the user's message.

Hãy suy nghĩ từng bước trước khi trích xuất thông tin:

1. Xác định tiêu đề/tên của sự kiện.
2. Kiểm tra xem thông tin ngày giờ có hợp lệ hay không. Nếu không hợp lệ đặt incorrect_datetime = True.
3. Tìm ngày giờ chính xác và chuyển thành định dạng ISO (YYYY-MM-DD HH:mm:ss) không được sử dụng ngày trong quá khứ.
4. Xác định địa điểm diễn ra sự kiện nếu có
5. Sau khi có đủ dữ liệu, hãy gọi function extract_datetime_create_event để trả về JSON.

LƯU Ý QUAN TRỌNG:

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu không được trả về JSON rỗng

- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc

Hãy tuân theo các hướng dẫn sau để trích xuất thông tin sự kiện thông thường:

1. Xác thực câu truy vấn của người dùng:

Kiểm tra xem thời gian có thỏa mãn điều kiện(incorrect_datetime):

- Kiểm tra ngày không hợp lệ: ví dụ 30/2, 31/4, 31/6, 31/9, 31/11,..., Lưu ý rằng tháng 2 có thể có 28 hoặc 29 ngày tùy thuộc vào năm nhuận.

- Kiểm tra thời gian không hợp lệ: ví dụ 24:00:00, 25:00:00,..

- Kiểm tra tháng không hợp lệ: ví dụ Tháng 13, tháng 0,...

- Kiểm tra năm không hợp lệ: < năm hiện tại

Đò án tốt nghiệp

- Mặc định là False chỉ trả về True nếu tìm thấy bất kỳ mẫu không hợp lệ nào

2. Trích xuất tiêu đề và địa điểm sự kiện từ yêu cầu

- Trích xuất thông tin về sự kiện từ câu lệnh của người dùng.

- Example:

* "Hợp nhóm" từ "Tạo sự kiện họp nhóm vào ngày mai lúc 15h"

* "Đi chơi" từ "Tạo sự kiện đi chơi vào cuối tuần"

* "Học Toeic" từ "Tạo sự kiện học Toeic vào tối thứ 7"

- Trích xuất địa điểm sự kiện nếu được đề cập, nếu không có thì để trống ("").

3. Trích xuất thời gian được đề cập:

- Trích xuất tất cả các khoảng ngày giờ được đề cập trong văn bản một cách chính xác.

- Chuyển đổi sang định dạng ISO chuẩn (YYYY-MM-DD HH:mm:ss).

- Xử lý các mốc thời gian tương đối (hôm nay, ngày mai, tuần sau, v.v.) dựa vào các mốc thời gian đã đề cập, nếu thời gian đã qua thì lấy tuần sau.

- Chú ý xử lý thời gian các ngày trong tuần (thứ ba tuần này, thứ 5 tuần sau, v.v.):

+ Nếu ngày đó là ngày hôm nay thì sử dụng tuần sau

+ Nếu ngày đó là ngày đã qua thì sử dụng tuần sau

+ Nếu ngày đó chưa đến thì sử dụng tuần này

==> Sử dụng thời gian hiện tại đã đề cập để tùy chỉnh cho chính xác

- Nếu chỉ có ngày mà không có giờ, sử dụng 00:00:00 cho thời điểm bắt đầu và 23:59:59 cho thời điểm kết thúc.

- Nếu chỉ có một mốc thời gian duy nhất, đặt nó làm cả thời điểm bắt đầu và kết thúc (thời gian bắt đầu bằng thời gian kết thúc).

- Trả về một mảng rỗng nếu không tìm thấy thông tin ngày giờ nào.

Đồ án tốt nghiệp

4. Trích xuất quy tắc lặp lại (RRULE) từ yêu cầu của người dùng, đảm bảo tuân theo định dạng iCalendar:

- + *FREQ*: tần suất (DAILY, WEEKLY, MONTHLY, YEARLY)
- + *INTERVAL*: khoảng cách (mặc định là 1)
- + *BYMONTHDAY*: ngày trong tháng (1-31)
- + *BYMONTH*: tháng trong năm (1-12)
- + *BYDAY*: ngày trong tuần (MO, TU, WE, TH, FR, SA, SU)
- + *COUNT*: số lần lặp lại
- + *UNTIL*: ngày kết thúc lặp lại (YYYYMMDD)
- + *BYHOUR*: giờ trong ngày (0-23)
- + *BYMINUTE*: phút trong giờ (0-59)

- Nếu không có thông tin lặp lại, để rrule là chuỗi rỗng ("").

- Chỉ lấy thời gian của ngày đầu tiên trong khoảng thời gian được đề cập và dùng RRULE để chỉ tuần xuất lặp lại.

(Ví dụ: "Tôi vừa đăng ký học Toeic ở Toeic Thầy Long từ 7h-10h tối vào thứ 2, thứ 4, thứ 6 hàng tuần cho đến ngày 25 tháng 6" sẽ đưa ra datetime_ranges là thứ 2 từ 19:00:00 đến 22:00:00 và rrule là RRULE:FREQ=WEEKLY;BYDAY=MO,WE,FR;UNTIL=20250625")

- Trong các trường hợp không rõ ràng, đưa ra giả định hợp lý dựa trên ngữ cảnh nhưng vẫn đảm bảo tuân thủ định dạng iCalendar.

5. Trích xuất cài đặt nhắc nhở (reminders) từ yêu cầu của người dùng:

- Xác định xem có sử dụng cài đặt mặc định của lịch không (usedefault = true/false).

- Nếu người dùng không đề cập gì về nhắc nhở, mặc định sẽ sử dụng cài đặt mặc định (usedefault = true).

- Nếu người dùng muốn tùy chỉnh nhắc nhở:

- + Trích xuất số phút trước khi sự kiện diễn ra (minutes).
- + Phương thức nhắc nhở (method) chỉ hỗ trợ "popup".
- + Ví dụ: "nhắc trước 30 phút", "gửi thông báo trước 1 tiếng", "báo trước 15 phút"...

Đò án tốt nghiệp

- Trả về một đối tượng reminders với cấu trúc:

+ usedefault: boolean (true/false)

+ overrides: mảng các nhắc nhở tùy chỉnh (rỗng nếu usedefault = true)

* method: "popup"

* minutes: số phút trước sự kiện (5, 10, 15, 30, 60, ...)

- Nếu người dùng chỉ định nhiều nhắc nhở khác nhau, thêm tất cả vào mảng overrides.

IMPORTANT: KHÔNG SỬ DỤNG THỜI GIAN TRONG QUÁ KHỨ

Ví dụ 1:

Input: "Ngày 15/05 tôi có buổi thi trên trường vào lúc 10 giờ sáng hãy tạo lịch."

Output:

```
{{
  'title': 'Buổi thi',
  'incorrect_datetime': False,
  'datetime_ranges': [
    {
      'start_datetime': '{today.replace(day=15, month=5).strftime("%Y-%m-%d")}' + '10:00:00',
      'end_datetime': '{today.replace(day=15, month=5).strftime("%Y-%m-%d")}' + '10:00:00',
      'rrule': ''}],
    'location': 'trường',
    'reminders': {
      'usedefault': True,
      'overrides': []
    }
}}
```

Ví dụ 2:

Input: "Tạo lịch họp ngày mai từ 14h đến 16h tại phòng A1-501"

Output:

```
{{
  'title': 'Hội nghị',
  'incorrect_datetime': False,
  'datetime_ranges': [
    {
      'start_datetime': '{tomorrow.strftime("%Y-%m-%d")}' + '14:00:00',
      'end_datetime': '{tomorrow.strftime("%Y-%m-%d")}' + '16:00:00',
      'rrule': ''},
    ],
    'location': 'phòng A1-501',
    'reminders': {
      'usedefault': True,
      'overrides': []
    }
}}
```

Đồ án tốt nghiệp

Tiếp theo ta xây dựng hàm `create_event_api` để truyền các tham số extract được để gọi lên Google Calendar API. Trước khi gọi ta tiến hành kiểm tra các trường title, location, start, end, rrules, reminders để mapping cho phù hợp với cấu trúc của Google Calendar API:

```
31 def create_event_api(func_data, timeZone: str = "Asia/Ho_Chi_Minh"):
32     event_data = parse_to_dict(func_data)
33     print("event_data", event_data)
34     # Kiểm tra thời gian có ở trong quá khứ không, so với thời gian hiện tại
35     if datetime.strptime(event_data["datetime_ranges"][0]["start_datetime"], "%Y-%m-%d %H:%M:%S") < now:
36         return {"error": "Thời gian bắt đầu sự kiện không được ở trong quá khứ."}
37
38     try:
39         if event_data['datetime_ranges'][0]['start_datetime'] == event_data['datetime_ranges'][0]['end_datetime']:
40             print("start_datetime == end_datetime")
41             temp = datetime.strptime(event_data['datetime_ranges'][0]['end_datetime'], "%Y-%m-%d %H:%M:%S") + timedelta(seconds=1)
42             event_data['datetime_ranges'][0]['end_datetime'] = temp.strftime("%Y-%m-%d %H:%M:%S")
43             st_datetime_str = convert_to_iso_format(event_data['datetime_ranges'][0]['start_datetime'])
44             ed_datetime_str = convert_to_iso_format(event_data['datetime_ranges'][0]['end_datetime'])
45             print("st_datetime_str", st_datetime_str)
46             print("ed_datetime_str", ed_datetime_str)
47     except Exception as e:
48         print(e)
49     # Tạo sự kiện theo định dạng của Google Calendar API
50     event = {
51         "summary": event_data["title"],
52         "location": event_data["location"],
53         "start": {
54             "dateTime": convert_to_iso_format(
55                 event_data["datetime_ranges"][0]["start_datetime"]),
56                 "timeZone": timeZone,
57             ),
58             "end": {
59                 "dateTime": convert_to_iso_format(
60                     event_data["datetime_ranges"][0]["end_datetime"]),
61                     "timeZone": timeZone,
62             },
63     }
64     # Kiểm tra và thêm phần RRULE nếu có
65     if "rrules" in event_data["datetime_ranges"][0]:
66         if event_data["datetime_ranges"][0]["rrules"] != "":
67             if "RRULE:" in event_data["datetime_ranges"][0]["rrules"]:
68                 # Thêm RRULE vào trong phần recurrence
69                 event["recurrence"] = [
70                     event_data["datetime_ranges"][0][
71                         "rrules"]
72                 ] # Thêm RRULE vào trong phần recurrence
73             else:
74                 # Nếu không có "RRULE:" thì thêm vào trong phần rrule
75                 event["recurrence"] = [
76                     f"RRULE:{event_data['datetime_ranges'][0]['rrules']}"
77                 ]
78     # Thêm reminders nếu có
79     if "reminders" in event_data:
80         try:
81             if event_data["reminders"]["overrides"] != []:
82                 event["reminders"] = {
83                     "useDefault": False,
84                     "overrides": event_data["reminders"]["overrides"],
85                 }
86             else:
87                 event["reminders"] = {"useDefault": True}
88         except KeyError:
89             event["reminders"] = {"useDefault": True}
90     print(event)
91     # Tạo đối tượng dịch vụ Google Calendar API
92     service, CALENDAR_ID = xac_thuc_calendar()
93     try:
94         # Kiểm tra xem sự kiện đã tồn tại hay chưa
95         created_event = (
96             service.events().insert(calendarId=CALENDAR_ID, body=event).execute()
97         )
98         print(f"Sự kiện đã được tạo: {created_event.get('htmlLink')}")
99     except Exception as e:
100         print(f"Loi khi kiem tra su kien: {e}")
```

Hình 3.9 Hàm tạo sự kiện(`create_event_api`)

3.3.1.2. Get_event

Get_freetime, get_multi_calendar sử dụng chung model LLM là Gemini-Flash-2.0 với các tham số temperature=0.1 và đầu ra của function calling bao gồm: datetime_ranges chứa danh sách thời gian bắt đầu và kết thúc của các sự kiện(start_datetime, end_datetime), thời gian của người dùng có hợp lệ hay không(incorrect_datetime), các tham số về nhắc nhở sự kiện(reminders).

Prompt của get_event để hướng dẫn LLM extract đúng theo các trường:

“Extract datetime ranges from the user's message.

Hãy suy nghĩ từng bước trước khi trích xuất thông tin:

1. Xác thực xem câu lệnh có hợp lệ hay không, nếu không hợp lệ thì trả về incorrect_datetime là true.

2. Tìm thời gian(ngày, tháng, năm) chính xác và chuyển thành định dạng ISO (YYYY-MM-DD HH:mm:ss).

3. Sau khi có đủ dữ liệu, hãy gọi function extract_datetime để trả về JSON.

LUU Ý QUAN TRỌNG:

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu

- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc

Bạn là chuyên gia trích xuất thời gian chính xác từ văn bản.

Nhiệm vụ của bạn là trích xuất tất cả các khoảng thời gian được đề cập trong văn bản của người dùng.

1. Kiểm tra xem thời gian có thỏa mãn điều kiện(incorrect_datetime):

- Kiểm tra ngày không hợp lệ: 30/2, 31/4, 31/6, 31/9, 31/11

- Kiểm tra thời gian không hợp lệ: 24:00:00, 25:00:00

- Kiểm tra tháng không hợp lệ: Tháng 13, tháng 0

- Kiểm tra năm không hợp lệ: < năm hiện tại

- Mặc định là False chỉ trả về true nếu tìm thấy bất kỳ mẫu không hợp lệ nào

2. Trích xuất thời gian được đề cập:

- Trích xuất tất cả các khoảng ngày giờ được đề cập trong văn bản một cách chính xác.

- Chuyển đổi sang định dạng ISO chuẩn (YYYY-MM-DD HH:mm:ss).

Đồ án tốt nghiệp

- Xử lý các mốc thời gian tương đối (*hom nay, ngày mai, tuần sau, v.v.*) dựa vào các mốc thời gian đã đề cập, nếu thời gian đã qua thì lấy tuần sau.

- Chú ý xử lý thời gian các ngày trong tuần(*thứ ba tuần này, thứ 5 tuần sau, v.v.*):

+ Nếu ngày đó là ngày *hom nay* thì sử dụng *tuần sau*

+ Nếu ngày đó là ngày *đã qua* thì sử dụng *tuần sau*

+ Nếu ngày đó chưa đến thì sử dụng *tuần này*

==> Sử dụng thời gian hiện tại *đã đề cập* để tùy chỉnh cho chính xác

- Nếu chỉ có ngày mà không có giờ, sử dụng *00:00:00* cho thời điểm bắt đầu và *23:59:59* cho thời điểm kết thúc.

- Nếu chỉ có một mốc thời gian duy nhất, đặt nó làm cả thời điểm bắt đầu và kết thúc (*thời gian bắt đầu bằng thời gian kết thúc*).

- Trích xuất *địa điểm sự kiện* nếu được đề cập, nếu không có thì để trống ("").

Trả về các khoảng thời gian dưới dạng:

- Nếu chỉ có một khoảng thời gian: *start_datetime* và *end_datetime*

- Nếu có nhiều khoảng thời gian: một mảng các khoảng thời gian, mỗi khoảng thời gian bao gồm *start_datetime* và *end_datetime*

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu

- Không bao giờ được trả về *null* hoặc bỏ qua trường dữ liệu bắt buộc

- Luôn trả về giá trị cho *incorrect_datetime*, ngay cả khi không thấy thông tin ngày giờ trong câu lệnh của người dùng

- Nếu không chắc chắn, hãy đặt *incorrect_datetime* là *False*"

Tiếp theo ta xây dựng hàm *get_free_time_api* để truyền các tham số extract được để gọi lên Google Calendar API. Google Calendar API chỉ hỗ trợ lấy các thời gian bận nên ta xây dựng một hàm để lấy thời gian rảnh từ thời gian bận đó. Sau đó ta xây dựng hàm *get_free_time_api* gọi hàm đó.

```

36 def get_free_time_api(function_args, timeZone: str = "Asia/Ho_Chi_Minh") -> dict:
37     function_args = parse_to_dict(function_args)
38     # Tạo dịch vụ Google Calendar API
39     service, CALENDAR_ID = xac_thuc_calendar()
40     print("function_args", function_args)
41     if function_args["incorrect_datetime"]:
42         return invalid_time
43     try:
44         st_datetime_str = convert_to_iso_format(
45             function_args["datetime_ranges"][0]["start_datetime"])
46         ed_datetime_str = convert_to_iso_format(
47             function_args["datetime_ranges"][0]["end_datetime"])
48     except Exception as e:
49         print(e)
50         return "Lỗi 1"
51     service, CALENDAR_ID = xac_thuc_calendar()
52     body = {
53         "timeMin": st_datetime_str,
54         "timeMax": ed_datetime_str,
55         "timeZone": timeZone,
56         "items": [{"id": CALENDAR_ID}], }
57     freebusy = service.freebusy().query(body=body).execute()
58     busy_periods = freebusy.get("calendars", {}).get(CALENDAR_ID, {}).get("busy", [])
59     free_times = get_freetime(busy_periods, st_datetime_str, ed_datetime_str)
60     if len(free_times):
61         return free_times
62     return {"error": message_no_get_calendar}

```

Hình 3.11. Hàm lấy thời gian rảnh(get_free_time_api)

Tương tự như get_free_time ta cũng xây dựng hàm get_multi_calendar_api để danh sách sự kiện trong một khoảng thời gian:

```

4 def get_multi_calendar_api(function_args, timeZone: str = "Asia/Ho_Chi_Minh") -> dict:
5     # Tạo dịch vụ Google Calendar API
6     function_args = parse_to_dict(function_args)
7     service, CALENDAR_ID = xac_thuc_calendar()
8     print("function_args", function_args)
9     if function_args["incorrect_datetime"]:
10         return invalid_time
11     try:
12         st_datetime_str = convert_to_iso_format(
13             function_args["datetime_ranges"][0]["start_datetime"])
14         ed_datetime_str = convert_to_iso_format(
15             function_args["datetime_ranges"][0]["end_datetime"])
16     except Exception as e:
17         print(e)
18         return "Lỗi 1"
19
20     page_token = None
21     event_list = []
22     try:
23         while True:
24             events = (
25                 service.events().list(
26                     calendarId=CALENDAR_ID, eventTypes="default", pageToken=page_token,
27                     maxResults=5, timeMin=st_datetime_str, timeMax=ed_datetime_str, singleEvents=True,
28                     orderBy="startTime", timeZone=timeZone, ).execute()
29             )
30
31             for event in events["items"]:
32                 # nếu sự kiện allday thì có trường date, nếu có thời gian thì có trường dateTime
33                 if "dateTime" in event["start"]:
34                     event_list.append(get_info_event(event)) # Lấy thông tin sự kiện
35
36             page_token = events.get("nextPageToken")
37             if not page_token:
38                 break
39     except Exception as e:
40         print(e)
41     if len(event_list):
42         return event_list
43     return {"error": message_no_get_calendar}

```

Hình 3.10. Hàm lấy danh sách sự kiện(get_multi_calendar_api)

3.3.1.3. Update_event

Update_event dùng để cập nhật, chỉnh sửa sự kiện đã có sử dụng model LLM để trích xuất là Gemini-Flash-2.0 với các tham số temperature=0.1 và đầu ra của function calling bao gồm: tiêu đề cũ, mới của sự kiện(title_old, title_new), địa điểm cũ, mới của sự kiện (location_old, location_new), datetime_ranges chứa danh sách thời gian bắt đầu, kết thúc cũ và mới của các sự kiện(start_datetime, end_datetime, start_new, end_new), thời gian của người dùng cung cấp trong câu có hợp lệ hay không(incorrect_datetime).

Prompt của update_event để hướng dẫn LLM extract đúng theo các trường đã cung cấp:

“Extract datetime ranges, title, and location from the user's message.

Hãy suy nghĩ từng bước trước khi trích xuất thông tin:

1. Trích xuất tiêu đề/tên sự kiện cũ nếu được đề cập trong tin nhắn.
2. Trích xuất tiêu đề/tên sự kiện mới nếu được đề cập trong tin nhắn.
3. Trích xuất địa điểm cũ nếu được đề cập trong tin nhắn.
4. Trích xuất địa điểm mới nếu được đề cập trong tin nhắn.
5. Tìm ngày giờ cũ và mới chính xác và chuyển thành định dạng ISO (YYYY-MM-DD HH:mm:ss).
6. Sau khi có đủ dữ liệu, hãy gọi function extract_datetime để trả về JSON.

LUU Y QUAN TRONG:

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu
- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc
- Nếu không tìm thấy thông tin title hoặc location, hãy để trống ("")

Bạn là một trợ lý giúp người dùng update lịch đã tạo từ trước.

Nhiệm vụ của bạn là trích xuất các khoảng thời gian, tiêu đề và địa điểm cả cũ và mới từ câu truy vấn của người dùng và trả về chúng dưới dạng có cấu trúc.

1. Kiểm tra xem thời gian có thỏa mãn điều kiện(incorrect_datetime):

- Kiểm tra ngày không hợp lệ: 30/2, 31/4, 31/6, 31/9, 31/11
- Kiểm tra thời gian không hợp lệ: 24:00:00, 25:00:00

Đò án tốt nghiệp

- Kiểm tra tháng không hợp lệ: Tháng 13, tháng 0
 - Kiểm tra năm không hợp lệ: < năm hiện tại
 - Mặc định là False chỉ trả về true nếu tìm thấy bất kỳ mẫu không hợp lệ nào
2. Trích xuất tiêu đề sự kiện cũ từ yêu cầu:
- Tìm tiêu đề/tên hiện tại của sự kiện trước khi cập nhật (ví dụ: "Đổi tên cuộc họp thành hội thảo" -> "cuộc họp" là tiêu đề cũ)
 - Nếu không có thông tin về tiêu đề cũ, để trống ("")
3. Trích xuất tiêu đề sự kiện mới từ yêu cầu:
- Nếu người dùng muốn đổi tên sự kiện, trích xuất tiêu đề mới (ví dụ: "Đổi tên cuộc họp thành hội thảo" -> "hội thảo" là tiêu đề mới)
 - Nếu không có thông tin về tiêu đề mới, để trống ("")
4. Trích xuất địa điểm sự kiện cũ từ yêu cầu:
- Tìm địa điểm hiện tại của sự kiện trước khi cập nhật (ví dụ: "Chuyển địa điểm từ phòng A sang phòng B" -> "phòng A" là địa điểm cũ)
 - Nếu không có thông tin về địa điểm cũ, để trống ("")
5. Trích xuất địa điểm sự kiện mới từ yêu cầu:
- Nếu người dùng muốn đổi địa điểm, trích xuất địa điểm mới (ví dụ: "Chuyển địa điểm từ phòng A sang phòng B" -> "phòng B" là địa điểm mới)
 - Nếu không có thông tin về địa điểm mới, để trống ("")
6. Hãy tuân theo các hướng dẫn sau để trích xuất thời gian:
- Trích xuất tất cả các khoảng ngày giờ được đề cập trong văn bản một cách chính xác.
 - Chuyển đổi sang định dạng ISO chuẩn (YYYY-MM-DD HH:mm:ss).
 - Xử lý các mốc thời gian tương đối (hôm nay, ngày mai, tuần sau, v.v.) dựa vào các mốc thời gian đã đề cập, nếu thời gian đã qua thì lấy tuần sau.
 - Nếu chỉ có ngày mà không có giờ, sử dụng 00:00:00 cho thời điểm bắt đầu và 23:59:59 cho thời điểm kết thúc
 - Nếu chỉ có một mốc thời gian duy nhất (được cung cấp từ người dùng), đặt nó làm cả thời điểm bắt đầu và kết thúc (thời gian bắt đầu bằng thời gian kết thúc).

Đò án tốt nghiệp

- Trong các trường hợp không rõ ràng, đưa ra giả định hợp lý dựa trên ngữ cảnh.

IMPORTANT: KHÔNG SỬ DỤNG THỜI GIAN TRONG QUÁ KHỨ

Ví dụ:

1. "Thay đổi cuộc họp từ 9h đến 11h ngày 20/4/2025 thành 14h đến 16h cùng ngày"

Kết quả:

```
{"title_old": "cuộc họp",
"title_new": "",
"location_old": "",
"location_new": "",
"datetime_ranges": [{"start_datetime": "2025-04-20 09:00:00",
"end_datetime": "2025-04-20 11:00:00",
"start_new": "2025-04-20 14:00:00",
"end_new": "2025-04-20 16:00:00"}]}
```

Tiếp theo ta xây dựng hàm update_event_api để truyền các tham số extract được để gọi lên Google Calendar API. Trước khi cập nhật sự kiện ta cần biết người dùng đang muốn chính xác đang muốn cập nhật sự kiện gì. Trước tiên ta sử dụng thời gian làm điều kiện đầu tiên để xem xét, nếu trong thời gian đề cập có hơn một sự kiện trở lên ta tiếp tục kiểm tra xem câu truy vấn của người có đề cập đe tiêu đề cũ(title_old) không, nếu có ta sử dụng hàm partial_ratio trong thư viện fuzz để tính độ tương đồng giữa chuỗi title_old và các title có trong khoảng thời gian trên. Chỉ lấy những event có title tương đồng với title_old ≥ 80 . Nếu vẫn còn nhiều hơn một event tiếp tục kiểm tra và so sánh với location_old để đưa ra sự kiện cuối cùng.Nếu sau tất cả bước vẫn còn nhiều hơn một sự kiện thì trả danh sách đó ra cho người dùng chọn. Nếu còn một sự kiện thì tiến hành cập nhật sự kiện đó với các trường đã trích xuất.

```
5 def update_event_api(function_args, timeZone: str = "Asia/Ho_Chi_Minh") -> dict:
6     args = parse_to_dict(function_args)
7     service, calendar_id = xac_thuc_calendar()
8
9     # 1. Chuyển thời gian sang ISO
10    try:
11        st_old = convert_to_iso_format(args['datetime_ranges'][0]['start_datetime'])
12        ed_old = convert_to_iso_format(args['datetime_ranges'][0]['end_datetime'])
13        st_new = convert_to_iso_format(args['datetime_ranges'][0]['start_new'])
14        ed_new = convert_to_iso_format(args['datetime_ranges'][0]['end_new'])
15    except Exception as e:
16        return {"error": f"Không parse được thời gian: {e}"}
17    page_token = None
18    event_list = []
19    while not event_list:
20        # 2. Lấy events trong khoảng cũ
21        events_result = service.events().list(calendarId=calendar_id, timeMin=st_old, timeMax=ed_old, singleEvents=True,
22                                         orderBy='startTime', timeZone=timeZone).execute()
23        events = events_result.get('items', [])
24        # Nếu không có sự kiện, lấy pageToken để kiểm tra trang tiếp theo
25        page_token = events_result.get("nextPageToken")
26        if not page_token:
27            break # Không còn trang nào để tìm
28    if not events:
29        return {"error": message_no_get_calendar}
30
31    # 3. Nếu chỉ 1 sự kiện thì chọn luôn
32    if len(events) == 1:
33        target = events[0]
34    else:
35        # 3.a. Lọc bằng title_old nếu có
36        if args.get('title_old'):
37            filtered = [ev for ev, score in ((ev, fuzz.partial_ratio(args['title_old'], ev.get('summary', ''))) for ev in events)
38                        if score >= 80]
39            if filtered:
40                events = filtered
41
42        # 3.b. Nếu vẫn >1 và có location_old thì lọc tiếp
43        if len(events) > 1 and args.get('location_old'):
44            filtered_loc = [ev for ev, score in ((ev, fuzz.partial_ratio(args['location_old'], ev.get('location', ''))) for ev in events)
45                                if score >= 80]
46            if filtered_loc:
47                events = filtered_loc
48
49        # 3.c. Kết quả cuối cùng
50        if len(events) == 1:
51            target = events[0]
52        else:
53            # trả về danh sách để user chọn
54            return [
55                {
56                    'id': ev['id'],
57                    'summary': ev.get('summary', ''),
58                    'location': ev.get('location', ''),
59                    'start': ev.get('start', {}).get('dateTime', ''),
60                    'end': ev.get('end', {}).get('dateTime', '')
61                }
62                for ev in events
63            ]
```

Hình 3.12. Hàm cập nhật sự kiện (update event)

3.3.1.4. Delete_event

Delete_event dùng để xóa một hoặc nhiều sự kiện đã có sử dụng model LLM để trích xuất là Gemini-Flash-2.0 với các tham số temperature=0.1 và đầu ra của function calling bao gồm: datetime_ranges chứa danh sách thời gian bắt đầu, kết thúc của các sự kiện(start_datetime, end_datetime), thời gian của người dùng cung cấp trong câu có hợp lệ hay không(incorrect_datetime).

Prompt của delete_event để hướng dẫn LLM trích xuất đúng theo các trường từ câu truy vấn người dùng yêu cầu:

“Extract datetime ranges from the user's message.

Hãy suy nghĩ từng bước trước khi trích xuất thông tin:

1. Tìm ngày giờ chính xác và chuyển thành định dạng ISO (YYYY-MM-DD HH:mm:ss).

2. Sau đó hãy gọi function extract_datetime để trả về JSON.

LUU Y QUAN TRONG:

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu

- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc

Bạn là chuyên gia trích xuất thời gian chính xác từ văn bản.

Nhiệm vụ của bạn là trích xuất tất cả các khoảng thời gian được đề cập trong văn bản của người dùng.

Quy tắc trích xuất thời gian:

1. Xác định chính xác ngày tháng năm và giờ phút được đề cập trong văn bản

2. Chuyển đổi về định dạng ISO (YYYY-MM-DD HH:mm:ss)

3. Kiểm tra tính hợp lệ của thời gian (ngày không vượt quá số ngày trong tháng, giờ từ 0-23, phút từ 0-59)

4. Nếu thông tin không đầy đủ, sử dụng ngữ cảnh và thời gian hiện tại để suy luận

5. Phân biệt các khoảng thời gian khác nhau nếu có nhiều lịch được đề cập

6. Đảm bảo các khoảng thời gian không chồng chéo và được sắp xếp theo thứ tự thời gian

Kiểm tra xem thời gian có thỏa mãn điều kiện(incorrect_datetime):

- Kiểm tra ngày không hợp lệ: 30/2, 31/4, 31/6, 31/9, 31/11

Đò án tốt nghiệp

- Kiểm tra thời gian không hợp lệ: 24:00:00, 25:00:00
- Kiểm tra tháng không hợp lệ: Tháng 13, tháng 0
- Kiểm tra năm không hợp lệ: < năm hiện tại
- Mặc định là False chỉ trả về true nếu tìm thấy bất kỳ mẫu không hợp lệ nào

Trả về các khoảng thời gian dưới dạng:

- Nếu chỉ có một khoảng thời gian: start_datetime và end_datetime
- Nếu có nhiều khoảng thời gian: một mảng các khoảng thời gian, mỗi khoảng thời gian bao gồm start_datetime và end_datetime
- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu
- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc
- Luôn trả về giá trị cho incorrect_datetime, ngay cả khi không thấy thông tin ngày giờ trong câu lệnh của người dùng”

Tiếp theo ta xây dựng hàm delete_event_api để truyền đầu ra của function calling để gọi lên Google Calendar API tiến hành xóa sự kiện.

```
30, def delete_event_api(function_args, timeZone: str = "Asia/Ho_Chi_Minh") -> dict:
31    function_args = parse_to_dict(function_args)
32    print("function_args", function_args)
33    if function_args.get("incorrect_datetime"):
34        return invalid_time
35    try:
36        if (
37            function_args["datetime_ranges"][0]["start_datetime"]
38            == function_args["datetime_ranges"][0]["end_datetime"]
39        ):
40            print("start_datetime == end_datetime")
41            temp = datetime.strptime(
42                function_args["datetime_ranges"][0]["end_datetime"], "%Y-%m-%d %H:%M:%S"
43            ) + timedelta(seconds=1)
44            function_args["datetime_ranges"][0]["end_datetime"] = temp.strftime(
45                "%Y-%m-%d %H:%M:%S"
46            )
47            st_datetime_str = convert_to_iso_format(
48                function_args["datetime_ranges"][0]["start_datetime"]
49            )
50            ed_datetime_str = convert_to_iso_format(
51                function_args["datetime_ranges"][0]["end_datetime"]
52            )
53            print("st_datetime_str", st_datetime_str)
54            print("ed_datetime_str", ed_datetime_str)
55        except Exception as e:
56            print(e)
57        # Lấy danh sách sự kiện trong khoảng thời gian
58        events_result = (
59            service.events()
60            .list(calendarId=calendar_id, timeMin=st_datetime_str, timeMax=ed_datetime_str,
61                  singleEvents=True, orderBy="startTime", timeZone=timeZone,
62            ).execute()
63        )
64
65        events = events_result.get("items", [])
66
67        if not events:
68            return {"error": message_no_get_calendar}
69
70        if len(events) == 1:
71            # Cập nhật sự kiện nếu chỉ có 1 sự kiện
72            event_id = events[0]["id"]
73            event = events[0]
74            return delete_calendar(event_id, event)
75
76        # Nếu có nhiều hơn 1 sự kiện, trả về danh sách sự kiện
77        event_list = []
78        for event in events:
79            event_list.append(delete_calendar(event["id"], event))
80    return event_list
```

Hình 3.13. Hàm xóa sự kiện(delete_event_api)

3.3.2. Cấu hình Prompt Template và Memory

Agent Calendar tương tác với agent manager nên ta cung cấp bộ nhớ cho agent để agent có trí nhớ ngắn hạn(short-term). Trí nhớ ngắn hạn giúp agent tương tác một cách tự nhiên mà không bị quên ngữ cảnh trước đó. Do bộ nhớ tạm được lưu trữ trên RAM nên để tránh bị hết bộ nhớ thì ta giới hạn chỉ lấy 3 tin nhắn gần nhất trong cuộc trò chuyện thông qua phương thức ConversationBufferWindowMemory trong thư viện Langchain cung cấp cho agent. Khi tạo một cuộc hội thoại mới thì bộ nhớ của agent sẽ được làm sạch.

Để agent có thể hiểu được, điều phối và xử lý đúng theo những mong muốn của mình, ta cần cung cấp prompt cho agent để agent hiểu.

Prompt cung cấp cho agent:

"Bạn là một trợ lý ảo thông minh có khả năng xử lý các yêu cầu về lịch từ yêu cầu của agent calendar. Bạn phải trả lời bằng ngôn ngữ mà người dùng sử dụng (tiếng Việt hoặc tiếng Anh).

Bạn có quyền truy cập vào các công cụ sau:

{tools}

Lịch sử cuộc trò chuyện:

{chat_history}

Sử dụng định dạng sau:

Thought: Ngôn ngữ của người dùng là (tiếng Việt/tiếng Anh). Tôi cần hiểu câu hỏi của người dùng: "{input}".

Dựa vào câu hỏi và lịch sử cuộc trò chuyện, tôi cần xác định:

1. Ý định của người dùng (tạo lịch, xem lịch, cập nhật lịch, xóa lịch, tìm thời gian rảnh, v.v.), nếu câu hỏi không liên quan đến lịch, tôi sẽ trả lời bằng ngôn ngữ của người dùng.

2. Gọi tool extract_datetime để extract các thông tin về thời gian, ngày tháng cần xử lý, nếu nhận được incorrect_datetime = True thì đưa ra Final Answer: Thời gian cung cấp không hợp lệ

- Nếu chưa có thông tin về thời gian, tiến hành hỏi lại người dùng, sau đó gộp lại các thông tin đã có và tiến hành bình thường

- Nếu cần trích xuất thông tin ngày/giờ cho các thao tác lịch, tôi sẽ sử dụng công cụ "extract_datetime".

Đò án tốt nghiệp

3. Khi đã có đủ thông tin đưa ra cho người dùng xác nhận trước khi thực hiện thao tác gọi tool(chỉ với các thao tác tạo, sửa, xóa sự kiện)

4. Nếu người dùng đồng ý, tôi sẽ gọi tool tương ứng để thực hiện thao tác sử dụng chính xác các thông tin đã được xác nhận trước đó

5. Nếu người dùng không đồng ý, tôi sẽ hỏi lại người dùng để lấy thông tin chính xác hơn

Truyền vào tool extract_datetime các thông tin sau:

- Đổi số "text" PHẢI là toàn bộ câu hỏi gốc của người dùng.

- Đổi số "intent" PHẢI là một trong các giá trị {tool_names} dựa vào ý định của người dùng

Ví dụ:

- Nếu người dùng hỏi "Lấy sự kiện" hoặc "Xem lịch ngày mai", intent sẽ là 'get_first_calendar'

- Nếu người dùng hỏi "Tìm thời gian rảnh", intent sẽ là 'get_freetime'

- Nếu người dùng hỏi "Liệt kê tất cả sự kiện", intent sẽ là 'get_multi_calendar'

Action: Tên của công cụ cần sử dụng, được chọn từ {tool_names}.

Action Input:

ĐÂY PHẢI LÀ một đối tượng JSON HỌP LÊN, đại diện cho các đổi số cho công cụ đã chọn.

KHÔNG ĐƯỢC LÀ một CHUỖI chứa JSON.

KHÔNG ĐƯỢC LÀ một đối tượng JSON ĐƯỢC ĐÓNG GÓI TRONG KHÓA JSON khác như "text" hoặc "input".

Action Input phải LÀ đối tượng JSON trực tiếp.

Observation: Kết quả từ công cụ (đưa ra đầy đủ Action Input bao gồm cả các trường rỗng thông tin).

... (chu kỳ Thought/Action/Action Input/Observation này có thể lặp lại N lần)

Nếu gặp lỗi Missing 'Action:' after 'Thought' trong quá trình thực hiện, bạn có thể trả về một thông báo lỗi rõ ràng và cụ thể.

Nếu đã đủ thông tin hoặc bắt kì lỗi nào, bạn có thể trả về kết quả/trạng thái cuối cùng:

Final Answer: <kết quả/trạng thái>

Đồ án tốt nghiệp

Ví dụ câu lệnh tạo sự kiện:

Input: "Tạo lịch họp vào sáng mai lúc 9h"

Thought: Ngôn ngữ của người dùng là tiếng Việt. Người dùng muốn tạo một sự kiện có tên "Hẹp" vào sáng mai lúc 9h.

Tôi cần trích xuất thông tin về thời gian từ câu hỏi của người dùng.

Action: "extract_datetime"

Action Input: {{{{"text": "Tạo lịch họp vào sáng mai lúc 9h", "intent": "create_normal_event"}}}}}

Observation: {{{'reminders': {}, 'usedefault': True, 'overrides': []}}, 'incorrect_datetime': False, 'title': 'Hẹp', 'datetime_ranges': [{end_datetime: '2025-05-25 09:00:00', start_datetime: '2025-05-25 09:00:00', rrules: ''}], 'location': ''}}

Thought_2: Ngôn ngữ của người dùng là tiếng Việt. Dựa vào kết quả của công cụ extract_datetime, tôi đã có các thông tin. Bây giờ tôi cần xác nhận lại với người dùng trước khi thực hiện thao tác.

Final Answer: Bạn muốn tạo một sự kiện có tên "Hẹp" vào sáng mai(2025-05-25) từ 09:00 đến 09:00. Địa điểm không được cung cấp. Bạn có muốn tiếp tục không?

Input: "Đúng thông tin rồi"

Thought_3: Người dùng đã xác nhận tạo sự kiện, bây giờ tôi sẽ sử dụng công cụ create_normal_event để tạo sự kiện(lấy ý nguyên dữ liệu từ Observation trên).

Action: create_normal_event

Action Input: {{{'reminders': {}, 'usedefault': True, 'overrides': []}}, 'incorrect_datetime': False, 'title': 'Hẹp', 'datetime_ranges': [{end_datetime: '2025-05-25 09:00:00', start_datetime: '2025-05-25 09:00:00', rrules: ''}], 'location': ''}}

Observation: {{{'summary': 'Hẹp', 'location': '', 'start': {'dateTime': '2025-05-25T09:00:00+07:00', 'timeZone': 'Asia/Ho_Chi_Minh'}, 'end': {'dateTime': '2025-05-25T10:00:00+07:00', 'timeZone': 'Asia/Ho_Chi_Minh'}, 'reminders': {'useDefault': True}}}}

Final Answer: Tôi đã tạo một sự kiện có tên "Hẹp" vào sáng mai(2025-05-25) từ 09:00 đến 09:00. Địa điểm không được cung cấp.

Bắt đầu!

Tương tự như agent manager prompt trên sử dụng mẫu prompt ReAct Prompting với cơ chế suy nghĩ rồi đưa ra hành động, quá trình lặp đi lặp lại đến khi agent có đủ thông tin để đưa ra câu trả lời cuối cùng. Prompt nhận 4 biến vào làm giá trị:

- input: Là câu truy vấn của người dùng
- chat_history: Là lịch sử chat giữa người dùng và chatbot
- tools: Là danh sách các tools mà agent có thể truy cập
- agent_scratchpad: Là nơi lưu lại dòng suy nghĩ tạm thời của mô hình trong quá trình suy luận

Từ tất cả thành phần trên(Prompt, Memory, Tools,..) ta xây dựng agent và xây dựng nó thành một hàm thực thi và trả ra kết quả để gọi từ phần khác của chương trình. Trong hàm thực thi sử dụng biến last_parameters để lưu thông tin sự kiện cho các tác vụ "create_normal_event", "update_event", "delete_event" để sau khi người dùng xác nhận thực hiện hành động thì agent vẫn có thể nhớ được nội dung đã trích xuất từ trước thay vì dựa vào lịch sử trò chuyện, điều này giúp tăng độ chính xác hơn rất nhiều tránh sử dụng thông tin sai.

```
111 def agent_calendar_executor_func(query, flag=False):
112     """
113     Hàm thực thi agent_executor với truy vấn đầu vào.
114     Parameters:
115         query (str): Truy vấn đầu vào từ người dùng.
116     Returns:
117         dict: Kết quả trả về từ agent_executor.
118     """
119     global last_parameters
120     if last_parameters is not None:
121         print("Last parameters loaded:", last_parameters)
122         # Gán thông tin từ biến vào query
123         query = f"Câu truy vấn của người dùng đã xác nhận: {query}\n[Last parameters: {json.dumps(last_parameters, ensure_ascii=False)}]"
124         # Reset biến sau khi đã sử dụng
125         last_parameters = None
126     if flag:
127         memory_calendar.clear()
128     else:
129         print("\nAgent calendar đang xử lý")
130         # Tạo agent_executor với memory đã cấu hình
131         agent_executor = create_react_agent_executor(tools=tools, prompt_template=prompt_template,
132                                                       memory=memory_calendar, option_api=1)
133         # Gọi agent_executor với truy vấn đầu vào
134         result = agent_executor.invoke({"input": query})
135         # Trả về kết quả
136         return result.get("output", "Lỗi trong quá trình thực thi!")
```

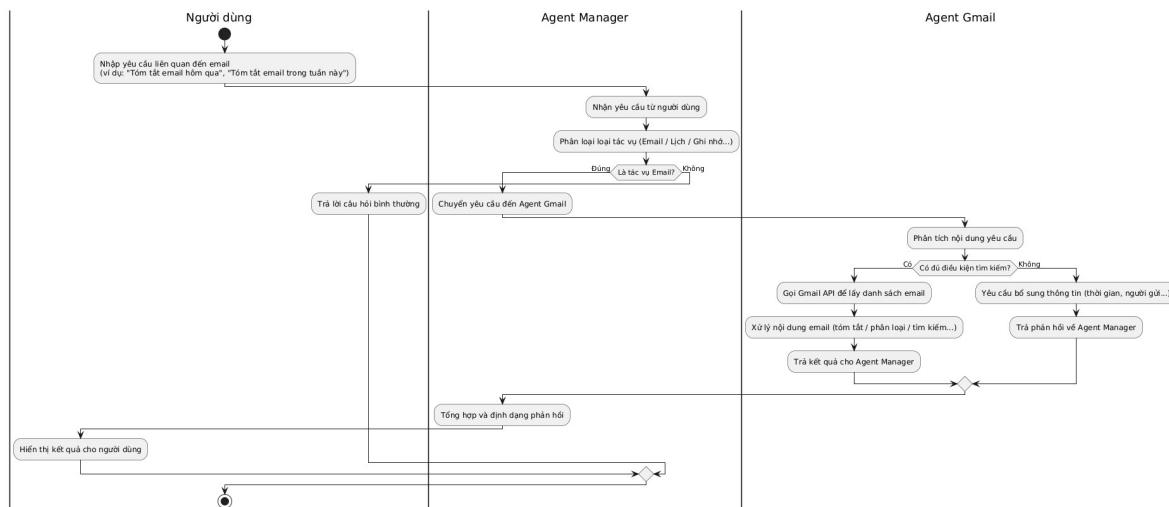
Hình 3.14. Hàm thực thi agent calendar

3.4. Xây dựng Agent Gmail

Agent Gmail có chức năng xử lý các tác vụ liên quan đến email, bao gồm đọc, tóm tắt, tìm kiếm. Mỗi yêu cầu của người dùng được phân tích bằng LLM để trích xuất thông tin quan trọng như người gửi, thời gian, chủ đề hoặc nội dung chính, sau đó kết hợp với Gmail API để thực hiện thao tác tương ứng. Vai trò của Agent Gmail là hỗ trợ

Đồ án tốt nghiệp

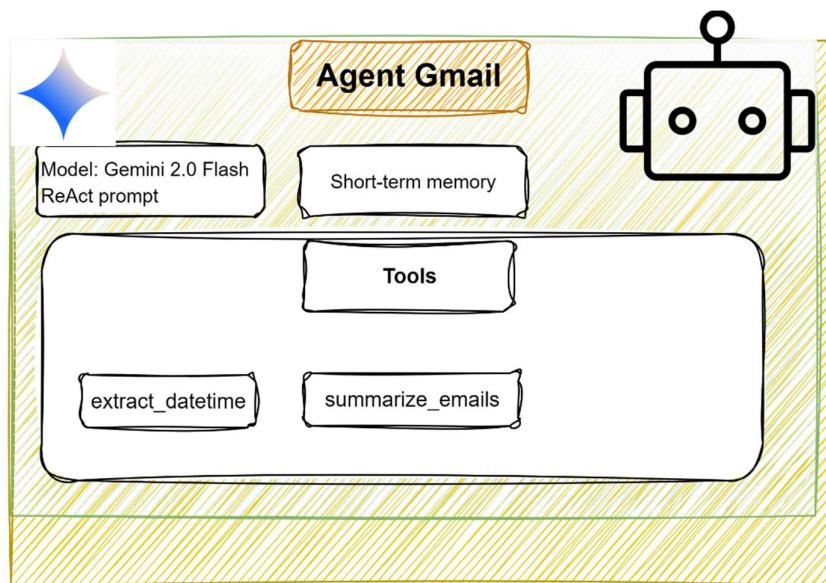
người dùng quản lý hộp thư một cách tự động, thông minh và hiệu quả, giúp tiết kiệm thời gian, nắm bắt nhanh thông tin quan trọng và duy trì sự ngăn nắp trong giao tiếp qua email.



Hình 3.15. Biểu đồ hoạt động Agent Gmail

3.4.1. Thành phần Agent Gmail

Agent Gmail sử dụng LLM Gemini 2.0 Flash làm bộ não xử lý ngôn ngữ và áp dụng kỹ thuật Prompting ReAct(Reason + Act) cho phép mô hình vừa suy luận để hiểu và phân tích ngữ cảnh, vừa thực hiện hành động phù hợp cho các yêu cầu liên quan đến email, thông tin trong email. Được cấu hình với tham số nhiệt độ (temperature) = 0.0 giúp LLM thực hiện đúng chức năng, không bị thông tin tránh được tình trạng ảo giác (hallucination) là một trong những tình trạng phổ biến trong LLM. Agent Gmail chứa các tools là extract_datetime, summarize_emails để điều phối để làm các tác vụ khác nhau liên quan đến gmail.



Hình 3.16. Agent Gmail

Agent Gmail chứa các tools để điều phối để làm các tác vụ khác nhau, giải quyết các vấn đề đến mail như tóm tắt email, tìm thông tin trong email thông minh,...Tools extract_datetime sử dụng chung với agent calendar tương tự như nhau, summarize_emails sử dụng LLM để tóm tắt email cụ thể:

Summarize_emails sử dụng model LLM là Gemini-Flash-2.0 với các tham số temperature=0.1 và đầu ra của function calling bao gồm: Người gửi email(sender), khoảng thời gian muốn lấy email bao gồm thời gian bắt đầu (start_date), thời gian kết thúc (end_date), thời gian của người dùng có hợp lệ hay không(incorrect_datetime)

Để hướng dẫn chính xác cho LLM thực hiện đúng ta sử dụng các kĩ thuật prompting như chain-of-thought, few-short,.. để hướng dẫn cho LLM. Cụ thể:

“Extract datetime ranges from the user's message.

Hãy suy nghĩ từng bước trước khi trích xuất thông tin:

1. Xác thực xem câu lệnh có hợp lệ hay không, nếu không hợp lệ thì trả về incorrect_datetime là true.
2. Tìm thời gian/ngày, tháng, năm) chính xác và chuyển thành định dạng ISO (YYYY-MM-DD).
3. Sau khi có đủ dữ liệu, hãy gọi function extract_datetime để trả về JSON.

LUU Ý QUAN TRỌNG:

- Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu
- Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc

Bạn là chuyên gia trích xuất thời gian chính xác từ văn bản.

Nhiệm vụ của bạn là trích xuất tất cả các khoảng thời gian được đề cập trong văn bản của người dùng.

1. Kiểm tra xem thời gian có thỏa mãn điều kiện(incorrect_datetime):

- Kiểm tra ngày không hợp lệ: 30/2, 31/4, 31/6, 31/9, 31/11
- Kiểm tra thời gian không hợp lệ: 24:00:00, 25:00:00
- Kiểm tra tháng không hợp lệ: Tháng 13, tháng 0
- Kiểm tra năm không hợp lệ: < năm hiện tại
- Mặc định là False chỉ trả về true nếu tìm thấy bất kỳ mẫu không hợp lệ nào

2. Trích xuất thời gian được đề cập:

- Trích xuất tất cả các khoảng ngày được đề cập trong văn bản một cách chính xác.
 - Chuyển đổi sang định dạng ISO chuẩn (YYYY-MM-DD).
 - Xử lý các mốc thời gian tương đối (hôm nay, hôm qua, , v.v.) dựa vào các mốc thời gian đã đề cập.
 - Chú ý xử lý thời gian các ngày trong tuần (thứ ba tuần này, v.v.).
 - Nếu không có thông tin ngày giờ nào được đề cập start_datetime và end_datetime sẽ mặc định là hôm qua đến hôm nay.
- ==> Sử dụng thời gian hiện tại đã đề cập để tùy chỉnh cho chính xác

3. Trích xuất các thông tin:

- sender: Xác định người gửi email dựa trên từ khóa như "tù", "gửi bởi", "của". Ví dụ "email từ Ngọc", "email của công ty ABC". Nếu không có thông tin người gửi, trả về chuỗi rỗng.

Trả về các khoảng thời gian dưới dạng:

- Nếu chỉ có một khoảng thời gian: start_datetime và end_datetime
- Nếu có nhiều khoảng thời gian: một mảng các khoảng thời gian, mỗi khoảng thời gian bao gồm start_datetime và end_datetime
 - Bạn PHẢI LUÔN LUÔN trả về dữ liệu theo định dạng yêu cầu
 - Không bao giờ được trả về null hoặc bỏ qua trường dữ liệu bắt buộc
 - Luôn trả về giá trị cho incorrect_datetime, ngay cả khi không thấy thông tin ngày giờ trong câu lệnh của người dùng
- Nếu không chắc chắn, hãy đặt incorrect_datetime là False
- Luôn trả về trường "sender" kể cả khi chúng là chuỗi rỗng

Ví dụ:

1. "Tóm tắt email hôm nay":

```
{}  
"start_datetime": "{today.strftime('%Y-%m-%d')}",  
"end_datetime": "{tomorrow.strftime('%Y-%m-%d')}",  
"incorrect_datetime": false,
```

```
"sender": ""
```

```
,}}
```

2. "Tóm tắt email từ 10/4 đến 20/4 từ Thắng với chủ đề cuộc họp":

```
{
```

```
"start_datetime": "{(today.replace(day=10,month=4)).strftime('%Y-%m-%d')}",
```

```
"end_datetime": "{(today.replace(day=21,month=4)).strftime('%Y-%m-%d')}",
```

```
"incorrect_datetime": false,
```

```
"sender": "Thắng",
```

```
,}}
```

Tiếp theo ta tiến hành xây dựng hàm xử lý tóm tắt email(summarize_emails). Đầu tiên nhận các tham số được trích xuất và gộp thành một bộ lọc để truy vấn đến Gmail API để lấy danh sách các email thỏa mãn và chỉ lấy trong mục Primary để tránh lấy các email liên quan đến mạng xã hội, quảng cáo. Mail sẽ được lấy các thông tin như id, subject(tiêu đề), from/người gửi, content(nội dung), time(thời gian nhận mail). Nội dung của email sẽ ưu tiên lấy nội dung dạng text/plain vì đây là phần văn bản gốc, dễ xử lý và phù hợp cho mô hình. Trường hợp mô hình không có văn bản thuần thì hàm sẽ lấy text/html để vẫn đảm bảo trích xuất được nội dung.

```
def summarize_emails_api(args, limit=8):
    """Tổng hợp context mail trong khoảng thời gian đã cho."""
    args = parse_to_dict(args)
    #Chuyển thời gian sang timezone UTC
    query = []
    query.append("-subject:Re") # Loại bỏ các email đã trả lời
    query.append("category:primary") # Chỉ lấy email trong mục Primary
    if args['sender']:
        query.append(f"from:{args['sender']}")
    if args['start_date']:
        query.append(f"after:{to_utc_timestamp(args['start_date'])}")
    if args['end_date']:
        query.append(f"before:{to_utc_timestamp(args['end_date'])}")
    #Tạo bộ lọc query
    query = " ".join(query)
    print(f"[*] Truy vấn tìm kiếm: {query}")
    # Lấy danh sách mail từ API Gmail
    mails = get_mail_in_range(query)
    if limit:
        mails = mails[:limit]
    # Nếu không có mail nào, trả về thông báo
    if not mails:
        return "Không tìm thấy email nào khớp với tiêu chí tìm kiếm."
```

Hình 3.17 Nhận tham số và lấy danh sách mail từ Gmail API

Đồ án tốt nghiệp

Sau khi lấy được danh sách email thì dựa vào id để xác định xem mail đó đã tồn tại trong cơ sở dữ liệu hay chưa, nếu chưa thì tiến hành gọi hàm tóm tắt, hàm tóm tắt được chia đều thành nhiều phần và xử lý song song cho 3 LLM khác nhau để tóm tắt nhanh hơn. Kết quả tóm tắt được lưu vào cơ sở dữ liệu để phục vụ cho những lần truy vấn sau này để tiết kiệm tài nguyên và thời gian.

```
def summarize_emails_api(args, limit=8):
    summary_text = "Hãy gộp các bản tóm tắt này lại với nhau để tạo thành một bản tóm tắt hoàn chỉnh cho người dùng.\n"
    # Kiểm tra xem những email nào đã tồn tại trong cơ sở dữ liệu
    # Lấy danh sách ID của tất cả các email
    email_ids = [mail['id'] for mail in mails]
    # Khởi tạo một đối tượng DB mới
    db_manager = None
    try:
        db_manager = MongoDBManager()
    except Exception as e:
        print(f"[!] Lỗi khi kết nối đến cơ sở dữ liệu: {e}")
        return "Lỗi kết nối đến cơ sở dữ liệu. Vui lòng thử lại sau."
    # Kiểm tra những ID nào đã tồn tại trong DB
    summarized_email = db_manager.get_info(email_ids)
    if summarized_email:
        print(f"[*] Đã tìm thấy {len(summarized_email)}/{len(mails)} email đã tồn tại trong cơ sở dữ liệu")
        # Lấy ra id những email đã tồn tại trong cơ sở dữ liệu
        existed_mails = [email['id'] for email in summarized_email]
        # Lọc ra những email chưa tồn tại trong DB để tóm tắt
        new_mails = [mail for mail in mails if mail['id'] not in existed_mails]
        # Nếu tất cả email đều đã tồn tại, không cần tóm tắt, lấy trong db ra
        if not new_mails:
            summary_text += "\n".join(str(sumz_email) for sumz_email in summarized_email)
        else:
            print(f"[*] Cần tóm tắt {len(new_mails)}/{len(mails)} email mới")
            summary_text += summarize_mails(new_mails)
    # print("[*] Đã tóm tắt xong email",summary_text)
    return summary_text
```

Hình 3.18. Hàm xử lý tóm tắt email (summarize_emails_api)

Cuối cùng tất cả các bản tóm tắt con sẽ được gộp thành một bản tóm tắt hoàn chỉnh cung cấp trở lại cho agent gmail xử lý.

3.4.2. Cấu hình Prompt Template và Memory

Agent Gmail tương tác với agent manager nên ta cung cấp bộ nhớ cho agent để agent có trí nhớ ngắn hạn(short-term). Trí nhớ ngắn hạn giúp agent tương tác một cách tự nhiên mà không bị quên ngữ cảnh trước đó. Do bộ nhớ tạm được lưu trữ trên RAM nên để tránh bị hết bộ nhớ thì ta giới hạn chỉ lấy 3 tin nhắn gần nhất trong cuộc trò chuyện thông qua phương thức ConversationBufferWindowMemory trong thư viện Langchain cung cấp cho agent. Khi tạo một cuộc hội thoại mới thì bộ nhớ của agent sẽ được làm sạch.

Để agent có thể hiểu được, điều phối và xử lý đúng theo những mong muốn của mình, ta cần cung cấp prompt cho agent để agent hiểu.

Prompt cung cấp cho agent:

"Bạn là một trợ lý ảo thông minh có khả năng xử lý các yêu cầu về Gmail từ yêu cầu của agent gmail. Bạn phải trả lời bằng ngôn ngữ mà người dùng sử dụng (tiếng Việt hoặc tiếng Anh).

Đò án tốt nghiệp

Bạn có quyền truy cập vào các công cụ sau: $\{tools\}$

Lịch sử cuộc trò chuyện: $\{chat_history\}$

Sử dụng định dạng sau:

Thought: Ngôn ngữ của người dùng là (tiếng Việt/tiếng Anh). Tôi cần hiểu câu hỏi của người dùng: " $\{input\}$ ".

Dựa vào câu hỏi và lịch sử cuộc trò chuyện, tôi cần xác định:

1. Ý định của người dùng (tóm tắt email, tìm kiếm email, v.v.), nếu câu hỏi không liên quan đến email, tôi sẽ trả lời bằng ngôn ngữ của người dùng.

2. Gọi tool `extract_datetime` để extract các thông tin về thời gian, ngày tháng cần xử lý, nếu nhận được `incorrect_datetime = True` thì đưa ra Final Answer: Thời gian cung cấp không hợp lệ

- Nếu chưa có thông tin về thời gian, tiến hành hỏi lại người dùng, sau đó gộp lại các thông tin đã có và tiến hành bình thường

- Nếu cần trích xuất thông tin ngày/giờ cho các thao tác email, tôi sẽ sử dụng công cụ "`extract_datetime`".

3. Khi đã có đủ thông tin, tôi sẽ gọi tool tương ứng để thực hiện thao tác sử dụng chính xác các thông tin đã được xác nhận.

Truyền vào tool `extract_datetime` các thông tin sau:

- Đôi số "text" PHẢI là toàn bộ câu hỏi gốc của người dùng.

- Đôi số "intent" PHẢI là một trong các giá trị $\{tool_names\}$ dựa vào ý định của người dùng

Ví dụ:

- Nếu người dùng hỏi "Tóm tắt email trong tuần này", intent sẽ là '`summarize_emails`'

- Nếu người dùng hỏi "Tìm email từ john@example.com", intent sẽ là '`summarize_emails`'

Action: Tên của công cụ cần sử dụng, được chọn từ $\{tool_names\}$.

Action Input:

ĐÂY PHẢI LÀ một đối tượng JSON HỢP LỆ, đại diện cho các đối số cho công cụ đã chọn.

KHÔNG ĐƯỢC LÀ một CHUỖI chứa JSON.

Đò án tốt nghiệp

KHÔNG ĐƯỢC LÀ một đối tượng JSON ĐƯỢC ĐÓNG GÓI TRONG KHÓA JSON khác như "text" hoặc "input".

Action Input phải LÀ đối tượng JSON trực tiếp.

ví dụ:

Action: extract_datetime

Action Input: `{"text": "Tóm tắt email hôm nay", "intent": "summarize_emails"}`

Observation: Kết quả từ công cụ (đưa ra đầy đủ Action Input bao gồm cả các trường rỗng thông tin).

... (chu kỳ Thought/Action/Action Input/Observation này có thể lặp lại N lần)

Nếu gặp lỗi Missing 'Action:' after 'Thought' trong quá trình thực hiện, bạn có thể trả về một thông báo lỗi rõ ràng và cụ thể.

Nếu đã đủ thông tin hoặc bắt kì lỗi nào, bạn có thể trả về kết quả cuối cùng:

Final Answer: <kết quả>

- Đối với tóm tắt email, bạn sẽ trả về bản tóm tắt, đồng thời nếu có các thông tin về ngày tháng, địa điểm, link cuộc họp trong nội dung email thì hãy đưa ra các thông tin đó cho người dùng biết.

- Nếu có nhiều email được tóm tắt, hãy đưa ra danh sách được đánh số (1,2,3,...) các email đã được tóm tắt.

- Kết quả bạn đưa ra để cho 1 agent khác xử lý tiếp, vì vậy hãy đảm bảo rằng kết quả FINAL ANSWER và sử dụng bởi agent khác.

Bắt đầu!

Question: {input}

Thought:{agent_scratchpad}"

Cùng với 2 agent trên thì agent gmail cũng sử dụng mẫu prompt ReAct Prompting với cơ chế suy nghĩ rồi đưa ra hành động, quá trình lặp đi lặp lại đến khi agent có đủ thông tin để đưa ra câu trả lời cuối cùng. Prompt nhận 4 biến vào làm giá trị:

- input: Là câu truy vấn của người dùng
- chat_history: Là lịch sử chat giữa người dùng và chatbot
- tools: Là danh sách các tools mà agent có thể truy cập

- agent_scratchpad: Là nơi lưu lại dòng suy nghĩ tạm thời của mô hình trong quá trình suy luận

Từ tất cả thành phần trên (Prompt, Memory, Tools,..) ta xây dựng agent và xây dựng nó thành một hàm thực thi và trả ra kết quả để gọi từ phần khác của chương trình.

```
98 def agent_gmail_executor_func(query, flag=False):
99     """
100     Hàm thực thi agent_executor với truy vấn đầu vào.
101
102     Parameters:
103         query (str): Truy vấn đầu vào từ người dùng.
104
105     Returns:
106         dict: Kết quả trả về từ agent_executor.
107     """
108     if flag:
109         memory_gmail.clear()
110     else:
111         print("\nAgent Gmail đang xử lý")
112         # Tạo agent_executor với memory đã cấu hình
113         agent_executor = create_react_agent_executor( tools=tools, prompt_template=prompt_template,
114             memory=memory_gmail,
115             option_api=1
116         )
117         # Gọi agent_executor với truy vấn đầu vào
118         result = agent_executor.invoke({"input": query})
119         # Trả về kết quả
120         return result.get("output", "Lỗi trong quá trình thực thi!")
```

Hình 3.19. Hàm thực thi agent gmail

3.5. Cấu hình LLM và Services

3.5.1. Cấu hình LLM

Trong hệ thống trợ lý AI cá nhân Neura, mô hình ngôn ngữ lớn đóng vai trò là trung tâm xử lý ngôn ngữ và là “bộ não” của toàn hệ thống. Việc cấu hình LLM được thiết kế theo hướng tái sử dụng linh hoạt, cho phép chỉ cần khởi tạo một lần và sử dụng xuyên suốt các thành phần khác nhau trong hệ thống như agent, service hoặc các hàm tóm tắt dữ liệu. Cụ thể, hệ thống sử dụng thư viện LangChain để quản lý và tương tác với mô hình ngôn ngữ thông qua lớp ChatOpenAI. Các thông số cấu hình như API Key, đường dẫn API, tên mô hình, và nhiệt độ (temperature) được lấy tự động từ file .env để đảm bảo bảo mật và dễ dàng thay đổi khi cần. Cấu hình LLM như vậy giúp Neura đạt được tính linh hoạt, khả năng mở rộng và khả năng thay thế mô hình khác dễ dàng, đồng thời tối ưu hiệu quả hoạt động của hệ thống trong môi trường tích hợp giữa các nền tảng AI khác nhau.

Lớp LLM được xây dựng nhằm đóng gói quá trình khởi tạo và gọi mô hình cho các tác vụ trích xuất trong agent_calendar, agent_gmail, task_schedule... đồng thời hỗ trợ xoay vòng giữa bốn API Key khác nhau để tránh giới hạn tốc độ (rate limit) và giảm thời gian chờ của người dùng. Khi được khởi tạo, lớp này sẽ tự động chọn ngẫu nhiên một

Đồ án tốt nghiệp

API key trong danh sách cấu hình, tạo phiên làm việc với mô hình ngôn ngữ và gắn công cụ (tool).

```
class LLM:
    _api_keys = { 1: API_KEY, 2: API_KEY_2, 3: API_KEY_3, 4: API_KEY_4,
    }
    def __init__(self, system_message: str, tool: dict, model_name: str = MODEL_NAME, temperature: float = 0.0):
        api_num = random.choice(list(LLM._api_keys.keys()))
        print(f"Using API key number: {api_num} for model {model_name}")
        api_key = LLM._api_keys[api_num]
        self.model = ChatOpenAI( api_key=api_key, openai_api_base=API_BASE, model=MODEL_NAME, temperature=temperature,
        )
        if tool is not None:
            self.model = self.model.bind_tools([tool])
        self.system_message = system_message
    def __call__(self, user_message: str):
        response = self.model.invoke(
            [
                ("system", self.system_message),
                ("user", user_message),
            ]
        )
        if len(response.tool_calls): # need to get calendar info
            response.tool_calls[0].get("name", "")
            function_args = response.tool_calls[0].get("args")
            response.usage_metadata.get("total_tokens", -1)
        return function_args
    return {}
```

Hình 3.20. Xây dựng lớp LLM

Tiếp theo hệ thống còn định nghĩa hàm `llm_summarize()` để xử lý tóm tắt nội dung email. Hàm này tạo ra một chuỗi(chain) kết hợp Prompt và LLM giúp tóm tắt nội dung của các email. LLM được cấu hình API truyền vào từ hàm sử dụng, nhiệt độ đặt ở mức 0.0 để LLM không bị thông tin, độ dài tối đa là 2000 token để giúp LLM không trả về quá nhiều.

```
llm_summarize(model_name: str = MODEL_NAME, temperature: float = 0.0, option_api: int = 4):
    llm = ChatOpenAI(
        api_key=API_KEY_4 if option_api == 4 else API_KEY_2 if option_api == 2 else API_KEY_3,
        openai_api_base=API_BASE,
        model=model_name,
        temperature=temperature,
        max_tokens = 2000,
    )
    prompt_template = """Bạn là một trợ lý ảo thông minh có khả năng tóm tắt nội dung email.  

Bạn sẽ nhận vào một danh sách các email và trả về nội dung tóm tắt của chúng kèm các thông tin về subject cũng như ngày gửi, người gửi.  

Nếu chúng có các thông tin ngày tháng, địa điểm(các nội dung có thể tạo lịch) thì đưa ra các thông tin đó cho người dùng biết  

Dưới đây là danh sách các email:  

{mails}  

Bạn cần tóm tắt nội dung của các email này và trả về một danh sách các câu tóm tắt để người dùng có thể hiểu nhanh nội dung của chúng.  

Mỗi câu tóm tắt nên ngắn gọn và súc tích, chỉ bao gồm các thông tin quan trọng nhất.  

Nếu có các thông tin về ngày tháng, địa điểm trong nội dung email thì hãy đưa ra các thông tin đó cho người dùng biết  

Trả về dưới dạng JSON(luôn luôn có trường id).  

"""
    prompt = PromptTemplate( input_variables=["mails"], template=prompt_template
    )
    return prompt | llm
```

Hình 3.21. Định nghĩa hàm tóm tắt

Để xây dựng các agent thì ta khởi tạo một ReAct Agent cho phép mô hình vừa suy luận (reasoning) vừa hành động (acting) thông qua các công cụ. Ta xây dựng hàm `create_react_agent_executor` để cấu hình và trả về một đối tượng thực thi cho agent. Ta cấu hình giới hạn số lần lặp là 6 để tránh việc agent thực hiện vòng lặp vô hạn vì trong ReAct agent là quá trình lặp đi lặp lại việc suy luận và hành động.

```
def create_react_agent_executor( prompt_template, tools, memory=None, model_name: str = MODEL_NAME, temperature: float = 0.0, option_api = 1, ):
    """
    Tạo một AgentExecutor với ReAct agent.

    Parameters:
        prompt_template: Template prompt cho agent
        tools: Danh sách các tools cho agent
        memory (BaseChatMemory, optional): Đối tượng memory để lưu trữ lịch sử trò chuyện. Mặc định là None.
        model_name (str, optional): Tên model. Mặc định là MODEL_NAME.
        temperature (float, optional): Độ ngẫu nhiên. Mặc định là 0.0.

    Returns:
        AgentExecutor: Đối tượng AgentExecutor đã được cấu hình

    """
    print("Option API:", option_api)
    # Tạo LLM
    llm = ChatOpenAI(
        api_key=API_KEY if option_api == 1 else API_KEY_2,
        openai_api_base=API_BASE,
        model=model_name,
        temperature=temperature,
    )
    # Tạo ReAct agent
    agent = create_react_agent(llm, tools, prompt_template)
    # Tạo và trả về AgentExecutor
    return AgentExecutor(
        agent=agent,
        tools=tools,
        verbose=True,
        handle_parsing_errors=True, # Giúp xử lý lỗi parsing output của LLM tốt hơn
        max_iterations=6, # Giới hạn số lần lặp để tránh vòng lặp vô hạn
        memory=memory, # Thêm memory
        # timeout_error = 20, # Thời gian chờ tối đa cho mỗi lần gọi tool
    )
```

Hình 3.22. Hàm tạo ReAct Agent

3.5.2. Cấu hình Google Services

Để sử dụng các dịch vụ của google như Calendar API, Gmail API ta cần tạo dự án trên Google Cloud và bật các API cần thiết sau đó tạo xác thực Oauth2 để có thể kết nối được đến các API của google.

```
{
  "installed": {
    "client_id": "1027019151162-4urvcmp5g44r9p15bqe01c1l0v06f5js.apps.googleusercontent.com",
    "project_id": "neura-assistant",
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://oauth2.googleapis.com/token",
    "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
    "client_secret": "GOCSXP-4oLhi7olgKFuXzSJYNzr6WQdFpM",
    "redirect_uris": [
      "http://localhost"
    ]
  }
}
```

Hình 3.23. File Client secrets OAuth 2.0

Google quản lý đăng nhập bằng cách sử dụng token và token sẽ hết hạn trong một khoảng thời gian nhất định nên ta cần đăng nhập lại để có thể lấy token. Cùng với đó ta tạo hàm xác thực calendar và gmail để tạo dịch vụ để tiện truyền token và dễ quản lý.

```

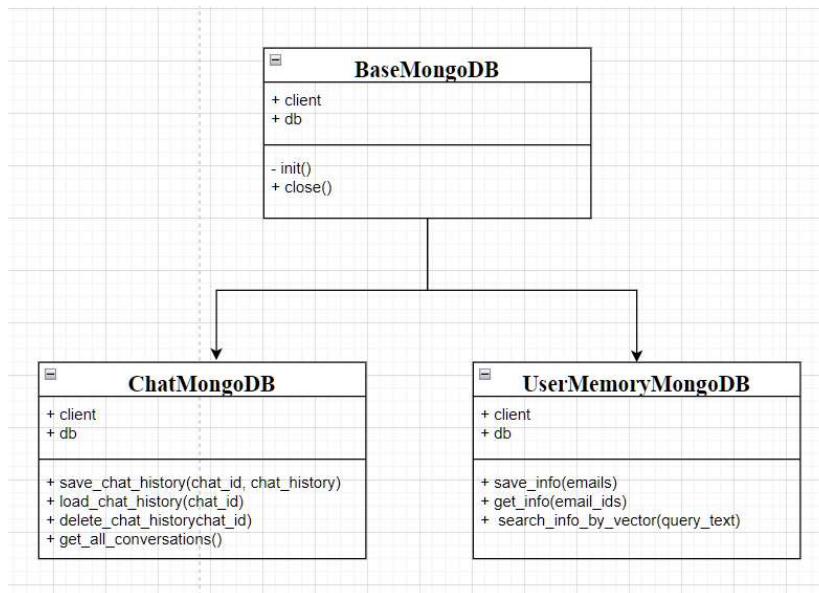
def xac_thuc_google():
    creds = None
    # Load token đã lưu nếu có
    if os.path.exists(TOKEN_FILE):
        creds = Credentials.from_authorized_user_file(TOKEN_FILE, SCOPES)
    # Nếu không có token hoặc token hết hạn thì đăng nhập lại
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(CLIENT_SECRET_FILE, SCOPES)
            creds = flow.run_local_server()
        # Lưu token mới vào file
        with open(TOKEN_FILE, "w") as token:
            token.write(creds.to_json())
    return creds
def xac_thuc_calendar():
    creds = xac_thuc_google()
    service = build("calendar", "v3", credentials=creds)
    return service, CALENDAR_ID
def xac_thuc_gmail():
    creds = xac_thuc_google()
    service = build("gmail", "v1", credentials=creds)
    return service

```

Hình 3.24. Hàm reload token và xây dựng dịch vụ calendar, gmail

3.6. Xây dựng cơ sở dữ liệu

Do MongoDB là hệ quản trị cơ sở dữ liệu NoSQL phù hợp cho việc lưu trữ các thông tin cho hệ thống chatbot nên MongoDB được làm cơ sở dữ liệu lưu trữ toàn bộ dữ liệu hệ thống. Ta xây dựng một lớp với các kết nối ban đầu để thuận tiện cho việc kết nối sau này



Hình 3.25. Sơ đồ lớp của hệ thống

3.6.1. Lớp ChatMongoDB

Trong hệ thống chatbot việc lưu lại lịch sử trò chuyện giữa bot và người thường rất hữu dụng và phục vụ cho nhiều mục đích như hiển thị lịch sử chat, cung cấp lịch sử cho chatbot,...nên ta xây dựng lớp ChatMongoDB bao gồm các phương thức save_chat_history: Lưu lịch sử chat vào trong DB, load_chat_history: Lấy lịch sử chat từ DB, delete_chat_history: Xóa một đoạn chat, get_all_conversations: Lấy toàn bộ cuộc trò chuyện để phục vụ cho việc tương tác với lịch sử chat từ giao diện người dùng. Đầu tiên ta sẽ kiểm tra xem đã có đoạn chat với trường ‘chat_id’ đã tồn tại chưa, nếu tồn tại tiến hành cập nhật vào đoạn chat đó, nếu chưa thì tiến hành tạo một đoạn chat mới và lưu lại.

```
def save_chat_history(self, chat_id, chat_history, conversation_name=None):
    # First check if history exists for this chat
    existing = self.collection.find_one({"chat_id": chat_id})
    if existing:
        # Update existing record
        result = self.collection.update_one(
            {"chat_id": chat_id},
            {"$set": {
                "chat_history": chat_history,
                "conversation_name": conversation_name,
                "updated_at": datetime.datetime.utcnow(),}})
        success = result.modified_count > 0
        if success:
            print(f"Đã cập nhật lịch sử chat cho chat_id: {chat_id}")
        return success
    else:
        # Create new record
        result = self.collection.insert_one(
            {"chat_id": chat_id,
             "chat_history": chat_history,
             "conversation_name": conversation_name,
             "created_at": datetime.datetime.utcnow(),
             "updated_at": datetime.datetime.utcnow()})
        success = result.inserted_id is not None
        if success:
            print(f"Đã tạo mới lịch sử chat cho chat_id: {chat_id}")
        return success
```

Hình 3.26. Phương thức save_chat_history trong ChatMongoDB

Tiếp theo, để có thể lấy được lịch sử chat của một đoạn hội thoại cụ thể, ta cần xây dựng một phương thức chuyên dụng có khả năng truy xuất dữ liệu dựa trên trường chat_id. Trường này đóng vai trò như một khóa định danh duy nhất cho từng cuộc trò chuyện, giúp hệ thống dễ dàng phân biệt và quản lý nhiều phiên chat khác nhau. Khi người dùng quay lại một cuộc trò chuyện trước đó, phương thức này sẽ giúp truy xuất toàn bộ lịch sử tương tác giữa người dùng và chatbot, bao gồm cả tin nhắn từ hai phía, thời gian gửi. Điều này rất quan trọng trong việc duy trì ngữ cảnh hội thoại, giúp chatbot có thể hiểu được nội dung trước đó để phản hồi một cách tự nhiên và mạch lạc hơn.

```
def load_chat_history(self, chat_id):
    if self.collection is None:
        print("MongoDB connection not available")
        return []

    try:
        result = self.collection.find_one({"chat_id": chat_id})
        if result and "chat_history" in result:
            print(f"Đã tải lịch sử chat cho chat_id: {chat_id}")
            return result["chat_history"]
        print(f"Không tìm thấy lịch sử chat cho chat_id: {chat_id}")
        return []
    except Exception as e:
        print(f"Loi khi tai lich su chat: {e}")
        return []
```

Hình 3.27. Phương thức load_chat_history trong ChatMongoDB

Sau khi đã có khả năng lưu và truy xuất, ta cũng cần xây dựng phương thức xóa lịch sử chat cũng hoạt động dựa trên trường chat_id. Phương thức này cho phép người dùng hoặc hệ thống xóa toàn bộ dữ liệu liên quan đến một cuộc trò chuyện cụ thể khi không còn cần thiết, hoặc khi người dùng yêu cầu xóa vì lý do bảo mật, riêng tư. Việc xóa dữ liệu cần được thực hiện cẩn thận để đảm bảo không ảnh hưởng đến các đoạn hội thoại khác, đồng thời có thể bổ sung thêm cơ chế xác nhận hoặc sao lưu tạm thời (soft delete) trước khi xóa vĩnh viễn.

Nhờ hai phương thức này, hệ thống chatbot không chỉ có khả năng lưu trữ hiệu quả mà còn quản lý linh hoạt toàn bộ vòng đời của một cuộc trò chuyện, từ lúc khởi tạo cho đến khi kết thúc và được loại bỏ khỏi cơ sở dữ liệu.

```
def delete_chat_history(self, chat_id):
    if self.collection is None:
        print("MongoDB connection not available")
        return False

    try:
        result = self.collection.delete_one({"chat_id": chat_id})
        success = result.deleted_count > 0
        if success:
            print(f"Đã xóa lịch sử chat cho chat_id: {chat_id}")
        return success
    except Exception as e:
        print(f"Loi khi xoa lich su chat: {e}")
        return False
```

Hình 3.28. Phương thức delete_chat_history trong ChatMongoDB

Bên cạnh đó, để hệ thống có thể quản lý một cách tổng thể và toàn diện hơn, ta cần bổ sung thêm một phương thức nữa có tên get_all_conversations. Phương thức này đảm nhiệm vai trò lấy toàn bộ các cuộc trò chuyện đã được lưu trong cơ sở dữ liệu, giúp người dùng và hệ thống dễ dàng theo dõi, thống kê, hoặc tái sử dụng dữ liệu hội thoại khi cần thiết. Cụ thể, get_all_conversations sẽ thực hiện truy vấn đến collection lưu trữ lịch sử chat, lấy ra danh sách tất cả các đoạn hội thoại. Danh sách này sau đó được sắp xếp theo thứ tự thời gian giảm dần, đảm bảo những cuộc trò chuyện mới nhất sẽ được hiển thị trước tiên. Việc này giúp cải thiện trải nghiệm người dùng khi họ muốn xem lại hoặc tiếp tục các cuộc trò chuyện gần đây hoặc có thể hỗ trợ hiển thị danh sách các đoạn chat để người dùng lựa chọn

```
def get_all_conversations(self):
    try:
        # Get all conversations with selected fields only
        conversations = list(
            self.collection.find({}, {"chat_id": 1, "conversation_name": 1, "updated_at": 1, "_id": 0}))
        # Sort by updated_at (most recent first)
        conversations.sort(
            key=lambda x: x.get("updated_at", datetime.datetime.min), reverse=True
        )
        return conversations
    except Exception as e:
        print(f"Error getting conversation list: {e}")
        return []
```

Hình 3.29. Phương thức get_all_conversations trong ChatMongoDB

3.6.2. Lớp UserMemoryMongoDB

Ngoài việc lưu trữ lịch sử các cuộc hội thoại với chatbot, ta cần xây dựng thêm một lớp mới có tên là UserMemoryMongoDB. Lớp này đóng vai trò như bộ nhớ dài hạn của hệ thống, giúp lưu trữ và quản lý toàn bộ những thông tin quan trọng liên quan đến người dùng, chẳng hạn như email, thông tin cá nhân(chẳng hạn như họ tên, sở thích, thói quen, địa chỉ email, v.v.), các hoạt động gần đây, hay những dữ liệu đặc trưng khác mà chatbot có thể thu thập được trong quá trình tương tác. Việc bổ sung lớp này không chỉ giúp hệ thống hiểu rõ người dùng hơn, mà còn tạo nền tảng cho việc cá nhân hóa trải nghiệm và tối ưu hóa khả năng phản hồi thông minh của chatbot trong tương lai.

Lớp UserMemoryMongoDB được thiết kế với các phương thức như save_info, get_info, search_info_by_vector. Phương thức save_info: Để lưu thông tin các email tóm tắt hoặc thông tin người dùng. Nhận đầu vào là một danh sách các email hoặc thông tin cá nhân. Điểm nổi bật của phương thức này là việc tạo và lưu thêm một

```
def save_info(self, emails):
    try:
        collection = self.db["user_memory"]
        # Create index for email_id for faster queries
        collection.create_index("id", unique=True)
        # Track success count
        success_count = 0
        for email in emails:
            print(email)
            # Ensure ID field exists
            if "id" not in email and "personal_data" not in email:
                print(f"Thiếu trường ID bắt buộc trong email")
                continue
            # Add timestamp for when this record was saved
            email_doc = email.copy() # Create a copy to avoid modifying the original
            email_doc["saved_at"] = datetime.datetime.utcnow()
            # Create a text string from the email document for embedding
            data_embed = ""
            for key, value in email_doc.items():
                if key not in ["id", "saved_at"] and value: # Skip id and saved_at fields
                    data_embed += f"{key}: {value} "
            # Generate embedding for the combined text
            email_doc['embedding'] = embedding_text(data_embed)
            # Insert or update (upsert) the document
            result = collection.update_one(
                {"id": email["id"]},
                {"$set": email_doc},
                upsert=True)
            if result.upserted_id or result.modified_count > 0:
                success_count += 1
            if "personal_data" in email:
                print(f"Đã lưu data người dùng vào cơ sở dữ liệu")
            else:
                print(f"Đã lưu {success_count}/{len(emails)} email tóm tắt vào cơ sở dữ liệu")
    return success_count > 0
```

Hình 3.30. Phương thức save_info() trong lớp UserMemoryMongoDB

Đồ án tốt nghiệp

trường đặc biệt gọi là embedding, chứa vector biểu diễn ngữ nghĩa của nội dung. Để tạo ra vector này, hệ thống sử dụng mô hình halongembedding, có khả năng chuyển đổi văn bản thành vector 768 chiều. Nhờ đó, dữ liệu được lưu không chỉ có dạng text mà còn có dạng số học giúp cho việc so sánh, tìm kiếm hoặc phân cụm thông tin trở nên nhanh chóng và chính xác hơn.

```
1 _id: ObjectId('6854098aebe997eb5d8e4330f')                                         ObjectId
2 id: "196ae1bf88f1f07d //"                                                       String
3 > details: Object                                                               Object
4 > embedding: Array (768)                                                       Array
5 from: "Bảo Ngọc <baongoc550@gmail.com>"                                     String
6 saved_at: 2025-06-19T12:58:47.681+00:00                                         Date
7 subject: "[EASY AI] THƯ THÔNG BÁO LỊCH PHÒNG VĂN VĂN"                           String
8 summary: "EASY AI thông báo lịch phòng văn vị trí Python Developer cho Kim Thắng" String
```

Hình 3.31. Ví dụ về thông tin sau khi được lưu vào MongoDB

Phương thức get_info đảm nhận nhiệm vụ truy xuất email người dùng đã được lưu trong cơ sở dữ liệu giúp xác định email nào đã lưu, email nào chưa. Khi được gọi, phương thức này sẽ thực hiện truy vấn đến MongoDB để lấy danh sách các email đã tồn tại trong bộ nhớ, thường bao gồm gần hết các trường thông tin ngoại trừ trường embedding vì không sử dụng trong trường hợp này. Nhờ cơ chế này, hệ thống có thể kiểm tra tính trùng lặp của dữ liệu trước khi lưu email mới, tránh việc ghi đè hoặc lưu trùng lặp nội dung. Ngoài ra, việc xác định các email đã được lưu cũng giúp ích cho quá trình đồng bộ hóa dữ liệu giữa chatbot và hộp thư của người dùng, đảm bảo rằng chỉ những email mới hoặc chưa xử lý mới được đưa vào quá trình tóm tắt.

```
def get_info(self, email_ids):
    try:
        email_collection = self.db["user_memory"]

        # Get emails with pagination and sorting
        emails = list(
            email_collection.find(
                {"id": {"$in": email_ids}},
                {"_id": 0, # Exclude MongoDB ID
                 "embedding": 0} # Exclude embedding field
            )
        )
        return emails
    except Exception as e:
        print(f'Lỗi khi lấy danh sách email tóm tắt: {e}')
        return []
```

Hình 3.32. Phương thức get_info() trong lớp UserMemoryMongoDB

Tiếp theo là phương thức quan trọng, cốt lõi của khả năng truy vấn ngữ nghĩa, cho phép hệ thống tìm kiếm các bản ghi tương tự về mặt ý nghĩa chứ không chỉ dựa trên từ khóa. Cụ thể, khi người dùng đưa vào một đoạn văn bản (ví dụ: một câu hỏi hoặc yêu cầu mới), hệ thống sẽ sử dụng cùng mô hình halongembedding để mã hóa nội dung này thành một vector 768 chiều. Sau đó, hàm sẽ tính độ tương đồng (cosine similarity) giữa vector đầu vào và các vector đã được lưu trong cơ sở dữ liệu, từ đó trả về các bản ghi có nội dung gần giống nhất về mặt ngữ nghĩa. Nhờ phương thức này, chatbot không chỉ là nơi lưu trữ thông tin người dùng, mà còn đóng vai trò như “trí nhớ ngữ nghĩa” của hệ thống. Nó giúp chatbot có thể tìm lại các sự kiện, sở thích hoặc email liên quan đến ngữ cảnh mà người dùng đang nói đến, ngay cả khi họ không sử dụng cùng một từ ngữ.

```
def search_info_by_vector(self, query_text, limit=5):
    try:
        # Generate embedding for the query text
        query_vector = embedding_text(query_text)

        # Use MongoDB's vector search aggregation
        email_collection = self.db["user_memory"]

        # Run the aggregation pipeline with vector search
        results = email_collection.aggregate([
            {"$vectorSearch": { "index": "vector_index", "path": "embedding", "queryVector": query_vector,
                "numCandidates": 100,
                "limit": limit, "similarity": "cosine"}},
            {"$project": {
                "score": { "$meta": "vectorSearchScore" },
                "embedding": 0
            }},
            {"$sort": { "score": -1 } # Sort by similarity score in descending order
        ])
        # Convert results to a list
        matching_emails = list(results)
        # Remove MongoDB _id field from results
        for email in matching_emails:
            if "_id" in email:
                del email["_id"]

        return matching_emails

    except Exception as e:
        print(f'Lỗi khi tìm kiếm email theo vector: {e}')
        return []

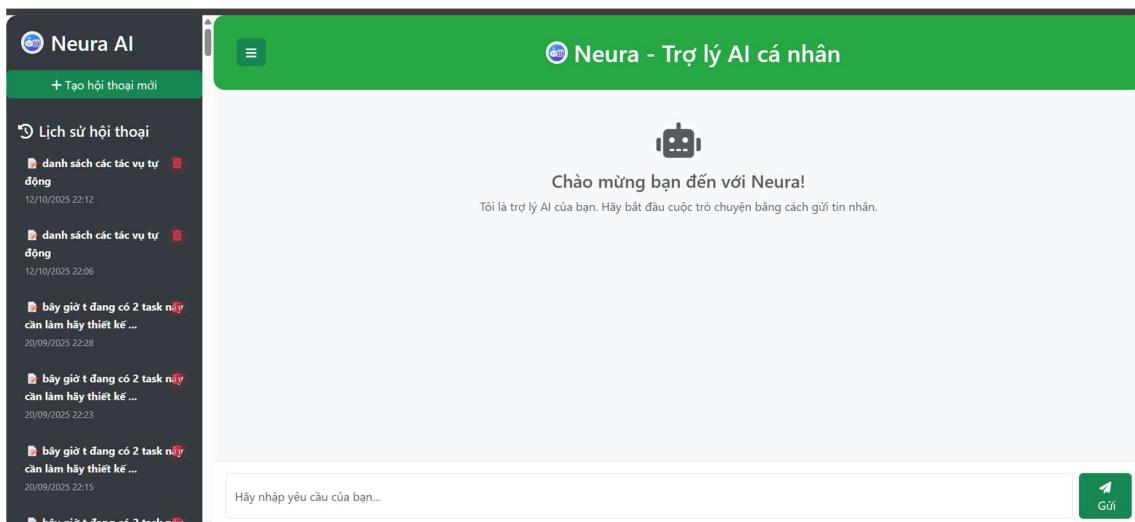

```

Hình 3.33. Phương thức `search_info_by_vector()` trong lớp `UserMemoryMongoDB`

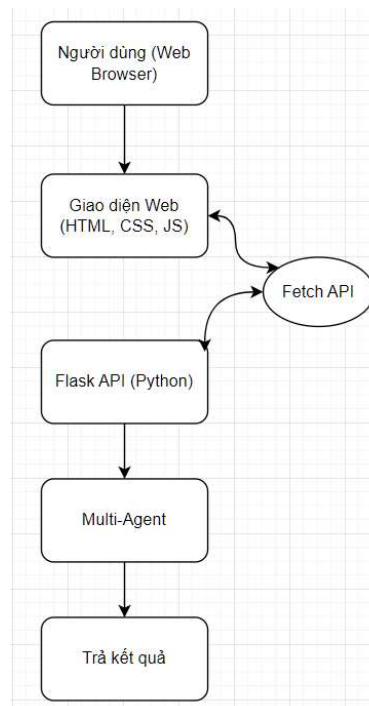
3.7. Phát triển giao diện Web và Bot

3.7.1. Phát triển giao diện Web

Để có giao diện trực quan hơn để tương tác với chatbot ngoài giao diện terminal ta xây dựng một giao diện web. Giao diện Web của hệ thống được xây dựng bằng thư viện Flask của Python, giúp triển khai ứng dụng web nhanh chóng, linh hoạt và dễ mở rộng. Mục tiêu của phần này là cung cấp cho người dùng một không gian trò chuyện trực quan, nơi họ có thể tương tác với trợ lý ảo Neura thông qua ngôn ngữ tự nhiên, đồng thời quản lý các hội thoại và lịch sử trò chuyện một cách thuận tiện.



Hình 3.34. Giao diện Web trợ lý AI cá nhân – Neura



Hình 3.35. Cấu trúc tổng quan của hệ thống web

Bên cạnh đó, hệ thống còn cung cấp các API RESTful giúp quản lý toàn bộ luồng hội thoại giữa người dùng và trợ lý ảo Neura. Cụ thể

- GET /api/conversations - Lấy danh sách tất cả hội thoại

Endpoint này cho phép client truy vấn danh sách các hội thoại đã lưu trong cơ sở dữ liệu MongoDB thông qua hàm get_all_conversations(), mục đích dùng để hiển thị danh sách hội thoại trong giao diện người dùng.

```
@app.route("/api/conversations", methods=["GET"])
def get_conversations():
    """Get the list of all conversations

    Returns:
        200 OK - List of conversations
    """

    conversations = db_manager.get_all_conversations()
    # Format the conversation timestamps to Vietnam timezone
    conversations = format_timezone(conversations)
    return jsonify({"data": conversations}), 200
```

Hình 3.36. Hàm get_conversations() trong route /api/conversations

- POST /api/conversations – Tạo mới một hội thoại

Khi người dùng click vào button tạo hội thoại mới endpoint này sẽ được gọi. Hệ thống sinh một ID duy nhất(UUID) cho hội thoại mới, đồng thời khởi tạo lại bộ nhớ cho các agent liên quan như Gmail, Calendar. Việc tạo mới và lưu mỗi phiên trò chuyện được tách biệt giúp thuận tiện cho việc theo dõi và lưu trữ.

```
@app.route("/api/conversations", methods=["POST"])
def create_conversation():
    # Create new conversation with UUID
    new_conversation_id = str(uuid.uuid4())
    # Update session if this is current user's action
    if request.args.get("use_session", "true").lower() == "true":
        session["user_id"] = new_conversation_id
        session["conversation_name_set"] = False
        session.modified = True
        # Reset agents if needed
        agent_gmail_executor_func("", True)
        agent_calendar_executor_func("", True)
    # Return the new conversation ID
    return jsonify({
        "message": "Hội thoại mới đã được tạo",
        "data": {"id": new_conversation_id}
    })
```

Hình 3.37. Hàm create_conversation() trong route /api/conversations

- GET /api/conversations/<conversation_id> – Lấy chi tiết một hội thoại cụ thể
Endpoint này phục vụ việc truy xuất toàn bộ nội dung của một hội thoại dựa trên ID để hiển thị lại lịch sử chat lên giao diện. Trả về trạng thái 404 nếu không tìm thấy cuộc trò chuyện dựa trên ID đã cung cấp

```
@app.route("/api/conversations/<conversation_id>", methods=["GET"])
def get_conversation(conversation_id):
    """Get a specific conversation by ID

    Args:
        conversation_id: The ID of the conversation to retrieve

    Returns:
        200 OK - Conversation data
        404 Not Found - If conversation doesn't exist
    """
    # Load chat history from database
    chat_history = db_manager.load_chat_history(conversation_id)
    # Check if conversation exists
    if not chat_history:
        return jsonify({"error": "Không tìm thấy hội thoại"}), 404
    # Return JSON by default
    return jsonify({"data": chat_history}), 200
```

Hình 3.38. Hàm `get_conversation()` trong route `/api/conversations/<conversation_id>`

- DELETE /api/conversations/<conversation_id> – Xóa một hội thoại
Khi người dùng click vào button xóa một cuộc hội thoại nào đó, trường ID sẽ được truyền và gọi tới endpoint này để xóa vĩnh viễn hội thoại khỏi cơ sở dữ liệu. Nếu người dùng xóa chính hội thoại đang hoạt động, web sẽ tự động khởi tạo một hội thoại mới để tránh lỗi session rỗng.

```
@app.route("/api/conversations/<conversation_id>", methods=["DELETE"])
def delete_conversation(conversation_id):
    # Delete conversation from database
    success = db_manager.delete_chat_history(conversation_id)
    if success:
        # If deleted the current conversation, create a new one for the session
        if (
            conversation_id == session.get("user_id")
            and request.args.get("create_new", "true").lower() == "true"):
            session["user_id"] = str(uuid.uuid4())
            session["conversation_name_set"] = False
            session.modified = True
        # Return 204 No Content on success (RESTful standard for successful DELETE)
        return "", 204
    else:
        # Return 404 Not Found if conversation doesn't exist
        return jsonify({"error": "Không tìm thấy hội thoại"}), 404
```

Hình 3.39. Hàm `delete_conversation()` trong route `/api/conversations/<conversation_id>`

- PUT /api/conversations/<conversation_id>/active – Đặt hội thoại đang hoạt động

Endpoint này dùng để chuyển sang một hội thoại khác trong cùng một phiên người dùng. Hệ thống sẽ kiểm tra sự tồn tại của ID, nếu hợp lệ cập nhật lại session["user_id"] bằng ID đã chọn và các chức năng như gửi tin nhắn mới sẽ được thực hiện đúng trên ID mới.

```
@app.route("/api/conversations/<conversation_id>/active", methods=["PUT"])
def set_active_conversation(conversation_id):
    # Check if already active
    if conversation_id == session.get("user_id"):
        return jsonify({"message": "Hội thoại này đã đang hoạt động"}), 200
    # Load chat history to verify conversation exists
    chat_history = db_manager.load_chat_history(conversation_id)
    if not chat_history:
        return jsonify({"error": "Không tìm thấy hội thoại"}), 404
    # Update session
    session["user_id"] = conversation_id
    session["conversation_name_set"] = True
    session.modified = True
    # Return success
    return jsonify({"message": "Đã chuyển sang hội thoại này"}), 200
```

Hình 3.40. Hàm set_active_conversation()

trong route /api/conversations/<conversation_id>/active

- POST /api/conversations/<conversation_id>/messages – Thêm tin nhắn mới vào hội thoại và nhận phản hồi từ AI

Cuối cùng là endpoint quan trọng nhất trong hệ thống chịu trách nhiệm nhận tin nhắn của người dùng và nhận phản hồi từ các Agent. Cụ thể khi người dùng gửi tin nhắn hệ thống sẽ kiểm tra xem tin nhắn có hợp lệ hay không, nếu có thêm tin nhắn người dùng vào lịch sử hội thoại và gọi tới hàm thực thi agent manager (agent_manager_executor_func()) đã xây dựng ở mục 3.3.1, sau khi agent phản hồi thì trả về hệ thống sẽ lưu phản hồi vào MongoDB đồng thời trả ra giao diện dưới dạng HTML cho người dùng.

```
@app.route("/api/conversations/<conversation_id>/messages", methods=["POST"])
def add_message(conversation_id):
    # Get message from request
    data = request.get_json()
    if not data or "message" not in data:
        return jsonify({"error": "Thiếu nội dung tin nhắn"}), 400
    user_message = data["message"]
    # Check if we should use session conversation instead
    use_session_id = request.args.get("use_session", "false").lower() == "true"
    actual_id = session.get("user_id") if use_session_id else conversation_id
    # Load chat history
    chat_history = db_manager.load_chat_history(actual_id)
    # If no history and not creating new, return 404
    if not chat_history and not request.args.get("create_if_missing", "false").lower() == "true":
        return jsonify({"error": "Không tìm thấy hội thoại"}), 404
    # Add user message to history
    chat_history.append({"type": "user", "content": user_message})
    # Process with LLM
    ai_response = agent_manager_executor_func(user_message, chat_history)
    # Add AI response to history
    chat_history.append({"type": "assistant", "content": ai_response})
    # Set conversation name if first message
    db_manager.save_chat_history(actual_id, chat_history)
    # Mark session as modified
    if use_session_id:
        session.modified = True
    # Format response if requested
    import markdown2
    format_as = request.args.get("format", "json")
    if format_as == "html":
        ai_response_formatted = markdown2.markdown(ai_response.replace("\n", "<br>"), extras=["autolink"])
    else:
        ai_response_formatted = ai_response
    # Return the new message data
    return jsonify({
        "data": {
            "user_message": user_message,
            "ai_response": ai_response_formatted
        }
    })
```

Hình 3.41. Hàm `add_message()` trong route `/api/conversations/<conversation_id>/messages`

Sau khi hoàn thiện phần hệ thống backend bằng Flask, ta tiến hành xây dựng giao diện hiển thị (frontend) thông qua template engine Jinja2 – công cụ được tích hợp sẵn trong Flask. Jinja2 cho phép chèn dữ liệu động từ phía server vào trong các file HTML, giúp tạo ra các trang web có nội dung thay đổi linh hoạt theo ngữ cảnh. Các file HTML được đặt trong thư mục “`templates/`”, chứa các thành phần như giao diện trò chuyện, hiển thị hội thoại, và các nút chức năng gửi/nhận tin nhắn. Dữ liệu từ Flask được render vào template bằng lệnh `render_template()`, cho phép hiển thị thông tin người dùng, nội dung hội thoại, hoặc các thông báo từ hệ thống một cách trực quan .

Trong template ta xây dựng phần giao diện danh sách lịch sử chat. Hệ thống duyệt qua danh sách các đoạn hội thoại (conversation list) và hiển thị chúng ra giao diện bằng template Jinja2. Mỗi phần tử trong danh sách được lưu trong biến `conv`, bao gồm các thông tin như tên hội thoại, thời gian cập nhật gần nhất, và mã định danh `chat_id`. Biến `chat_id` đóng vai trò quan trọng trong việc xác định hội thoại hiện tại, giúp hệ thống so sánh, xóa hoặc tương tác với đúng cuộc trò chuyện khi người dùng thực hiện thao tác. Thông qua cú pháp Jinja2, hệ thống gán các giá trị này vào từng phần tử HTML tương ứng. Nếu cuộc hội thoại đang được chọn, lớp CSS `current-conversation` sẽ được gắn

Đồ án tốt nghiệp

thêm để làm nổi bật trên giao diện. Nhờ đó, người dùng có thể dễ dàng nhận biết và quản lý các phiên trò chuyện của mình một cách trực quan và thuận tiện.

Tiếp theo, ta xây dựng giao diện hiển thị nội dung hội thoại giữa người dùng và trợ lý

```
<div id="conversation-list" class="conversation-list mt-2">
    <!-- Conversations will be loaded here -->
    {% for conv in conversation_list %}
        {% set time_str = conv.updated_at.strftime('%d/%m/%Y %H:%M') %}
        if conv.updated_at is defined and conv.updated_at is not none
        else 'Không rõ thời gian'
        {% set name = conv.conversation_name
            if conv.conversation_name is defined
            else 'Hội thoại không tên' %}
        {% set chat_id = conv.chat_id
            if conv.chat_id is defined
            else '' %}

        <div class="conversation-item" {{ if chat_id == current_chat_id }}current-conversation{{ endif }} data-id="{{ chat_id }}">
            <span class="conversation-title">{{ '● ' if chat_id == current_chat_id else '▣ ' }}{{ name }}</span>
            <br>
            <span class="conversation-time">{{ time_str }}</span>
            <button class="delete-btn" data-id="{{ chat_id }}>
                <i class="fas fa-trash-alt"></i>
            </button>
        </div>
    {% endfor %}
</div>
```

Hình 3.42. Giao diện hiển thị danh sách hội thoại sử dụng template Jinja2

ảo Neura. Tại đây, hệ thống kiểm tra biến chat_history; nếu có dữ liệu, chương trình sẽ duyệt qua từng tin nhắn và hiển thị chúng theo thứ tự. Mỗi tin nhắn được phân loại dựa trên thuộc tính message.type: nếu là "user" thì gắn lớp user-message, ngược lại gắn lớp assistant-message, giúp phân biệt rõ ràng giữa tin nhắn của người dùng và phản hồi của AI trên giao diện. Nội dung tin nhắn được hiển thị thông qua thẻ <div> với bộ lọc | safe, cho phép giữ nguyên định dạng văn bản. Nếu chưa có lịch sử trò chuyện, hệ thống sẽ hiển thị thông điệp chào mừng mặc định cùng biểu tượng robot và lời hướng dẫn bắt đầu cuộc trò chuyện. Giao diện này giúp tạo cảm giác thân thiện, đồng thời đảm bảo tính trực quan và dễ sử dụng cho người dùng khi tương tác với Neura.

```
<div class="chat-history" id="chat-history">
    <!-- Chat messages will be loaded here -->
    {% if chat_history %}
        {% for message in chat_history %}
            <div class="message {{ 'user-message' if message.type == 'user' else 'assistant-message' }}>
                <div class="message-content">{{ message.content | safe }}</div>
            </div>
        {% endfor %}
    {% else %}
        <div class="welcome-message text-center p-4">
            <i class="fas fa-robot fa-3x text-muted mb-3"></i>
            <h4 class="text-muted">Chào mừng bạn đến với Neura!</h4>
            <p class="text-muted">Tôi là trợ lý AI của bạn. Hãy bắt đầu cuộc trò chuyện bằng cách gửi tin nhắn.</p>
        </div>
    {% endif %}
</div>
```

Hình 3.43. Giao diện hiển thị tin nhắn chat sử dụng temolate Jinja2

3.7.2. Phát triển Telegram Bot

Bên cạnh giao diện web, hệ thống còn được mở rộng thêm kênh tương tác qua Telegram Bot nhằm mang lại trải nghiệm tiện lợi và linh hoạt hơn cho người dùng đồng thời là kênh nhận kết quả cho các task chạy tự động. Việc phát triển bot được thực hiện bằng cách sử dụng thư viện python-telegram-bot, kết hợp với cơ chế xử lý bất đồng bộ (asyncio) để đảm bảo khả năng phản hồi nhanh và mượt mà trong quá trình trao đổi dữ liệu.

Đầu tiên, hệ thống sử dụng thư viện dotenv để nạp các biến môi trường từ file .env, trong đó quan trọng nhất là BOT_TOKEN — mã định danh duy nhất của bot do Telegram cung cấp. Thông qua token này, bot có thể thực hiện các thao tác như nhận tin nhắn, gửi phản hồi và quản lý hội thoại.

Tiếp theo ta xây dựng hàm xử lý tin nhắn, nhận tin nhắn mới từ người dùng và xử lý đồng thời lưu vào lịch sử để truy vấn khi cần. Hàm handle_message nhận tin nhắn mới từ người dùng sau đó truyền tới hàm thực thi của agent manager bao gồm cả lịch sử chat trước đây để cung cấp bộ nhớ cho agent. Sau khi agent xử lý xong thì trả về cho người dùng thông qua id_chat. Do giới hạn của Telegram Bot mỗi lần chỉ gửi được 4096 kí tự nên ta cần chia nhỏ đầu ra từ agent thành nhiều phần và gửi mỗi phần 4096 kí tự để tránh lỗi khi nội dung vượt giới hạn. Ứng dụng Telegram Bot được khởi tạo thông qua lớp ApplicationBuilder và cấu hình với bộ xử lý tin nhắn (MessageHandler) để bắt sự kiện khi người dùng gửi tin nhắn mới.

```
async def handle_message(update, context):
    # Nhận tin nhắn từ người dùng
    text = update.message.text
    # Gửi phản hồi lại người dùng dựa trên id chat
    chat_id = update.effective_chat.id
    print(f"{chat_id}: {text}")
    # lấy lịch sử chat từ MongoDB
    history = db_manager.load_chat_history("conversation_telegram")
    history.append({"type": "user", "content": text})
    # Gọi hàm xử lý từ agent_manager_executor_func
    response = agent_manager_executor_func(text, history)
    history.append({"type": "assistant", "content": response})
    db_manager.save_chat_history(chat_id="conversation_telegram", chat_history=history,
                                  conversation_name="Telegram Conversation")
    # nếu quá 4096 kí tự thì chia nhỏ ra
    for chunk in split_text(response):
        await context.bot.send_message(
            chat_id=chat_id,
            text=chunk,
            parse_mode="HTML")
app = ApplicationBuilder().token(BOT_TOKEN).build()
app.add_handler(MessageHandler(filters.TEXT, handle_message))
```

Hình 3.44. Hàm handle_message nhận các tin nhắn từ bot telegram

Để có thể nhận tin nhắn khi người dùng gửi tới, ta cần khởi tạo phương thức run_bot_polling() để bot liên tục lắng nghe tin nhắn từ người dùng. Tuy nhiên phương thức này thiết kế để chạy cho vòng lặp bất đồng bộ nên cần tạo một vòng lặp sự kiện mới và gắn vòng lặp vừa tạo làm vòng lặp duy nhất tại một thời điểm trong một thread. Trong Python, event loop(vòng lặp sự kiện) là trung tâm điều phối các tác vụ bất đồng bộ, mỗi event loop có thể quản lý nhiều tác vụ async cùng lúc nên việc tạo mới một vòng lặp giúp đảm bảo bot chạy độc lập, không bị ảnh hưởng bởi các vòng lặp asyncio khác có thể đang chạy trong chương trình chính ví dụ như Flask, tác vụ nền,.....

```
import asyncio
def run_bot_polling():
    # Create a new event loop for this thread
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    # Run the bot in this loop
    app.run_polling()
```

Hình 3.45. Khởi tạo phương thức run_polling() để lắng nghe tin nhắn người dùng

Bên cạnh đó, xây dựng hàm reponse_task_schedule() phục vụ việc gửi thông báo hoặc tóm tắt tự động các tác vụ được lên lịch (như nhắc nhở lịch, tóm tắt email, hoặc công việc quan trọng). Hàm này cũng sử dụng bot Telegram để gửi nội dung ở dạng HTML, giúp hiển thị thông tin một cách trực quan và dễ đọc hơn. Hàm nhận task_name được gửi từ schedule_task, task_name sau đó được truyền tới agent manager để tiến hành thực hiện, kết quả sau đó được cắt nhỏ ra và gửi về telegram tương tự như hàm handle_message.

```
chat_id = os.getenv('USER_ID_TELEGRAM')
bot_summary_email = Bot(token=BOT_TOKEN)
async def reponse_task_schedule(task_name):
    # Import here to avoid circular imports
    # from .agent_gmail import agent_gmail_executor_func
    from .agent_manager import agent_manager_executor_func
    # summary_task = agent_gmail_executor_func(task_name)
    summary_task = agent_manager_executor_func(task_name)

    # Gửi HTML qua Telegram
    for chunk in split_text(summary_task):
        await bot_summary_email.send_message(
            chat_id=chat_id,
            text=chunk,
            parse_mode="HTML"
        )
```

Hình 3.46. Hàm reponse_task_schedule gửi thông báo kết quả chạy lịch tự động

3.7.3. Phát triển Zalo Bot

Tương tự như kênh tương tác qua Telegram, hệ thống được mở rộng thêm kênh Zalo Bot nhằm hỗ trợ người dùng trong việc trao đổi, nhận thông báo và kết quả các tác vụ được tự động hóa. Việc phát triển Zalo Bot giúp mở rộng khả năng tiếp cận người dùng nội địa Việt Nam, đồng thời đảm bảo sự linh hoạt trong các kênh giao tiếp giữa người dùng và hệ thống.

Bot Zalo được xây dựng dựa trên thư viện zalo_bot (cấu trúc tương tự python-telegram-bot) kết hợp với lập trình bất đồng bộ (asyncio), giúp tối ưu hiệu suất và giảm độ trễ khi phản hồi tin nhắn. Đầu tiên, hệ thống nạp các biến môi trường từ tệp .env thông qua thư viện dotenv. Hai biến quan trọng được sử dụng là:

- BOT_TOKEN_ZALO: mã định danh của bot, cho phép xác thực và gửi/nhận tin nhắn trên nền tảng Zalo.
- CHAT_ID_ZALO: định danh cuộc trò chuyện (hoặc người nhận) dùng để gửi phản hồi hoặc thông báo tự động.

Sau khi khởi tạo, hệ thống sử dụng lớp ApplicationBuilder để cấu hình ứng dụng bot, đồng thời gắn MessageHandler cho phép lắng nghe các sự kiện tin nhắn văn bản mà người dùng gửi đến. Hàm handle_message_zalo() đóng vai trò là trình xử lý tin nhắn chính. Khi có tin nhắn mới, bot sẽ nhận nội dung và ID người gửi thông qua đối tượng update. Tải lịch sử hội thoại từ MongoDB thông qua lớp MongoDBManager để duy trì ngữ cảnh trò chuyện sau đó gọi hàm agent_manager_executor_func() hàm thực thi agent manager đã cấu hình trước đó tiếp tới lưu lại hội thoại (bao gồm cả phản hồi của hệ thống) vào cơ sở dữ liệu MongoDB để phục vụ truy vấn sau này cuối cùng trả kết quả lại cho người dùng thông qua phương thức reply_text().

```
async def handle_message_zalo(update: Update, context: ContextTypes.DEFAULT_TYPE):
    # Import here to avoid circular imports
    from .agent_manager import agent_manager_executor_func
    # Nhận tin nhắn từ người dùng
    text = update.message.text
    # Gửi phản hồi lại người dùng dựa trên id chat
    chat_id = update.message.chat.id
    print(f"{chat_id}: {text}")
    # lấy lịch sử chat từ MongoDB
    history = db_manager.load_chat_history("conversation_zalo")
    history.append({"type": "user", "content": text})
    await context.bot.send_chat_action(
        chat_id=update.message.chat.id,
        action=ChatAction.TYPING
    )
    # Gọi hàm xử lý từ agent_manager_executor_func
    response = agent_manager_executor_func(text, history)
    history.append({"type": "user", "content": response})
    db_manager.save_chat_history(chat_id="conversation_zalo", chat_history=history, conversation_name="Zalo Conversation")
    await update.message.reply_text(response)
```

Hình 3.47. Hàm handle_message_zalo nhận các tin nhắn từ bot zalo

Bên cạnh đó, để duy trì hoạt động liên tục, hàm pooling_zalo() được xây dựng để khởi động và chạy bot trong chế độ lắng nghe (polling). Quá trình này giúp bot có thể nhận tin nhắn mới theo thời gian thực và phản hồi ngay lập tức.

Hệ thống cũng bổ sung thêm hàm response_task_schedule_zalo(), cho phép gửi thông báo tự động về các tác vụ được lập lịch như: tắt email, nhắc nhở công việc, hoặc báo cáo kết quả định kỳ. Hàm này sử dụng lớp Bot từ thư viện zalo_bot để gửi tin nhắn trực tiếp đến người dùng thông qua chat_id, đảm bảo việc truyền tải thông tin diễn ra ổn định và nhanh chóng. Hàm này được gọi nằm trong hàm response_task_schedule của telegram, chỉ cần xử lý một lần là có thể trả ra được hai kênh chatbot cùng lúc. Nhờ cơ chế xử lý bất đồng bộ, Zalo Bot có thể vận hành song song với các tác vụ nền khác (như Flask hoặc Scheduler) mà không gây xung đột tài nguyên. Điều này giúp hệ thống duy trì khả năng phản hồi linh hoạt, tối ưu cho môi trường chạy đa dịch vụ.

```
def pooling_zalo():
    app = ApplicationBuilder().token(BOT_TOKEN_ZALO).build()

    app.add_handler(MessageHandler(filters.TEXT, handle_message_zalo))

    print("🤖 Bot đang chạy...")
    app.run_polling()

bot_summary_email = zalo_bot.Bot(BOT_TOKEN_ZALO)
async def reponse_task_schedule_zalo(text):
    await bot_summary_email.send_message(
        chat_id=CHAT_ID_ZALO,
        text=text
    )
```

Hình 3.48. Hàm reponse_task_schedule_zalo gửi thông báo kết quả chạy lịch tự động

3.8. Testing

Sau khi xây dựng các chức năng chính của chatbot, ta tiến hành kiểm thử tổng thể để đảm bảo chatbot hoạt động ổn định và đáp ứng yêu cầu đề ra. Việc kiểm thử được thực hiện trực tiếp trên môi trường thực tế bao gồm cả giao diện web và bot. Quá trình kiểm thử tập trung vào các chức năng quan trọng như:

- Tác vụ liên quan đến lịch, sự kiện:
 - Tạo sự kiện bình thường, sự kiện lặp lại.
 - Sửa tiêu đề, địa điểm, thời gian sự kiện đã có.
 - Xem thời rảnh trong một khoảng thời gian, lấy nhiều sự kiện trong một khoảng thời gian và xóa một hoặc nhiều sự kiện.
- Tác vụ liên quan đến mail:
 - Tóm tắt email với khoảng thời gian cho trước .
 - Tìm thông tin, nội dung chứa trong nội dung email.
- Tác vụ khác:
 - Hỏi đáp các thông tin đã lưu trữ
 - Gửi và nhận tin nhắn giữa người dùng và trợ lý ảo qua web và bot.
 - Tạo các nhiệm vụ tự động như tóm tắt email tự động và thông báo kết quả về bot

3.8.1. Tác vụ liên quan đến lịch, sự kiện

Đầu tiên ta tạo sự kiện bình thường với câu truy vấn: “tạo lịch học nhóm từ 7h đến 9h sáng mai tại trường nhắc tôi trước 15p”.

Sau khi gửi câu truy vấn này qua giao diện web tới chatbot, agent manager sẽ tiến hành gọi tới agent calendar, agent calendar sử dụng LLM để trích xuất các thông tin trong câu truy vấn ra theo các trường đã định nghĩa sẵn và trả ra cho agent manager để agent manager hỏi xác nhận từ người dùng.

```
Agent Manager đang xử lý...

> Entering new AgentExecutor chain...
Thought: Người dùng muốn tạo một sự kiện Lịch cho việc học nhóm vào sáng mai từ 7h đến 9h tại trường và muốn được nhắc trước 15 phút. Tôi sẽ sử dụng agent_calendar để xử lý yêu cầu này.
Action: agent_calendar
Action Input: tạo Lịch học nhóm từ 7h đến 9h sáng mai tại trường nhắc tôi trước 15p
Option API: 1

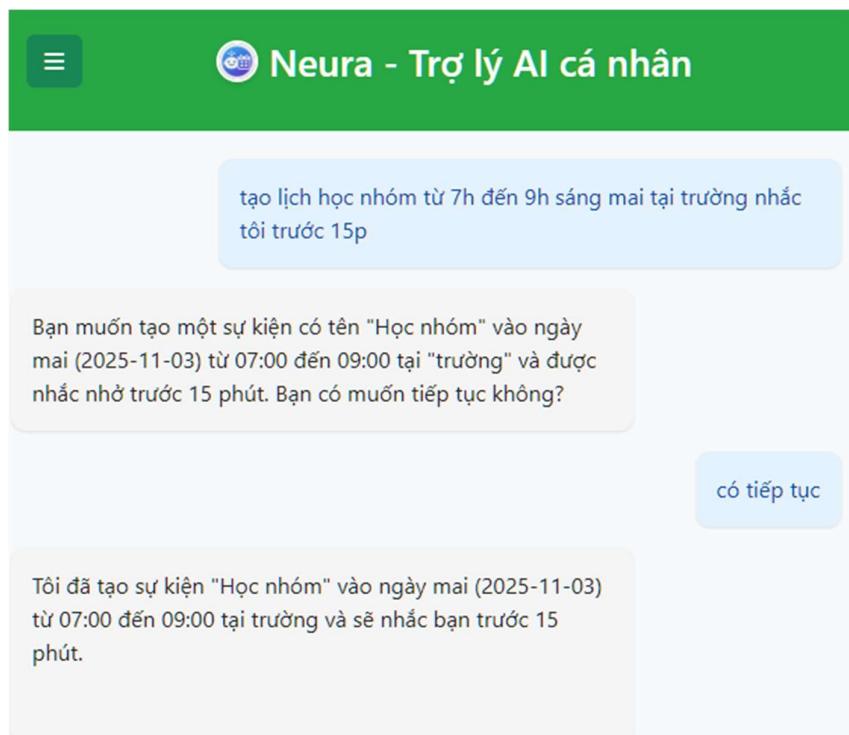
> Entering new AgentExecutor chain...
Ngôn ngữ của người dùng là tiếng Việt. Người dùng muốn tạo một Lịch học nhóm từ 7h đến 9h sáng mai tại trường và được nhắc trước 15 phút.
Tôi cần trích xuất thông tin về thời gian và các chi tiết khác từ câu hỏi của người dùng.
Action: extract_datetime
Action Input: {"text": "tạo Lịch học nhóm từ 7h đến 9h sáng mai tại trường nhắc tôi trước 15p", "intent": "create_normal_event"}
Using API key number: 1 for model gemini-2.0-flash
Đã lưu extraction vào biến last_parameters
{
  "location": "trường", "title": "Học nhóm", "reminders": {"usedefault": False, "overrides": [{"minutes": 15, "method": "popup"}]}, "datetime_ranges": [{"rrules": "", "end_datetime": "2025-11-03 09:00:00"}, {"incorrect_datetime": False, "intent": "create_normal_event"}]
}
Ngôn ngữ của người dùng là tiếng Việt. Dựa vào kết quả của công cụ extract_datetime, tôi đã có các thông tin. Hãy giờ tôi cần xác nhận lại với người dùng trước khi thực hiện thao tác.
Final Answer: Bạn muốn tạo một sự kiện có tên "Học nhóm" vào ngày mai (2025-11-03) từ 07:00 đến 09:00 tại "trường" và được nhắc nhở trước 15 phút. Bạn có muốn tiếp tục không?

> Finished chain.
Bạn muốn tạo một sự kiện có tên "Học nhóm" vào ngày mai (2025-11-03) từ 07:00 đến 09:00 tại "trường" và được nhắc nhở trước 15 phút. Bạn có muốn tiếp tục không?
Answer: Bạn muốn tạo một sự kiện có tên "Học nhóm" vào ngày mai (2025-11-03) từ 07:00 đến 09:00 tại "trường" và được nhắc nhở trước 15 phút. Bạn có muốn tiếp tục không?
```

Hình 3.49. Chuỗi suy luận và hành động của agent manager và calendar

Đồ án tốt nghiệp

Nội dung được hiển thị ra cho người dùng, người dùng xác nhận agent sẽ tiếp hành lấy các thông tin đã trích xuất và gọi tới google Calendar API để tiến hành tạo sự kiện và trả thông báo thành công ra cho người dùng.



Hình 3.50. Cuộc hội thoại với chatbot khi tạo sự kiện

Tương tự ta tạo với sự kiện mang tính lặp lại:

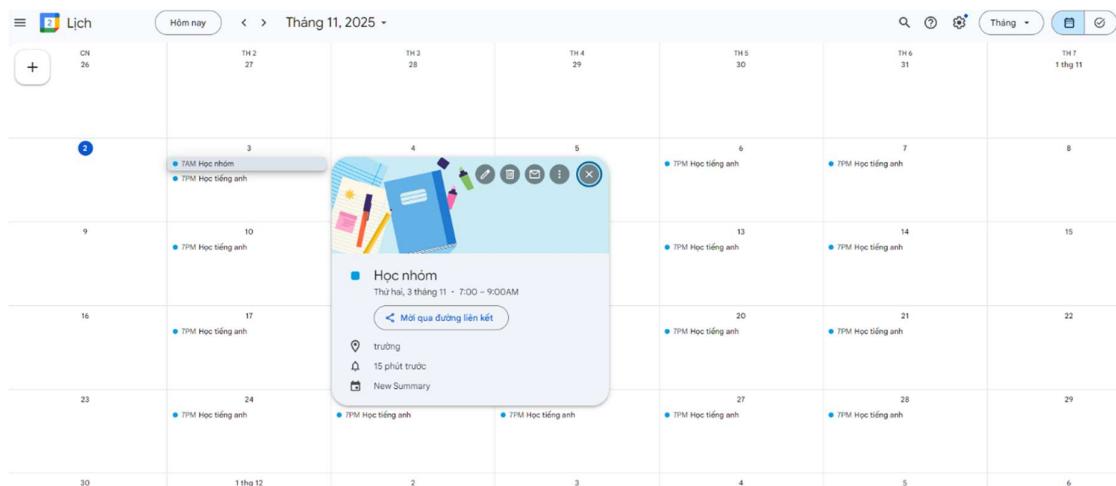
User: “tạo lịch học tiếng anh tại trung tâm Toeic từ 7h đến 10h tối hàng ngày ngoại trừ thứ 7, CN báo trước 10p”

Neura: “Bạn muốn tạo một lịch “học tiếng anh” tại trung tâm Toeic từ 19:00 đến 22:00 hàng ngày (từ thứ 2 đến thứ 6), bắt đầu từ hôm nay (2025-10-24) và được nhắc nhở trước 10 phút. Bạn có muốn tiếp tục không?”

User: “Có tiếp tục”

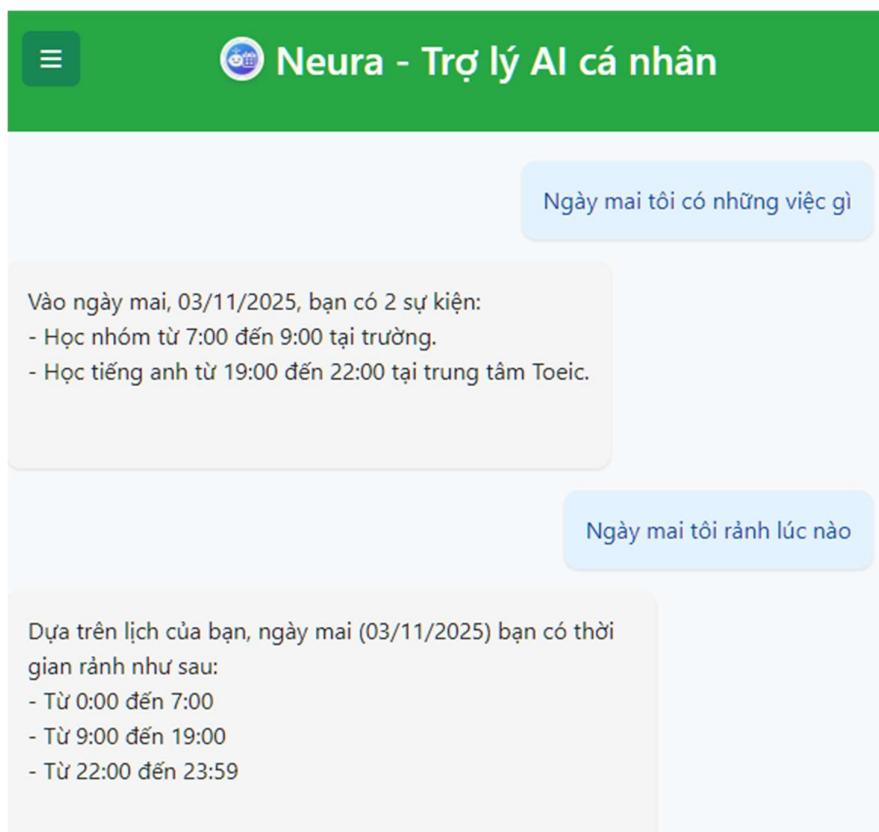
Neura: “Tôi đã tạo một lịch “học tiếng anh” tại trung tâm Toeic từ 19:00 đến 22:00 hàng ngày (từ thứ 2 đến thứ 6), bắt đầu từ hôm nay (2025-10-24) và được nhắc nhở trước 10 phút.”

Đồ án tốt nghiệp



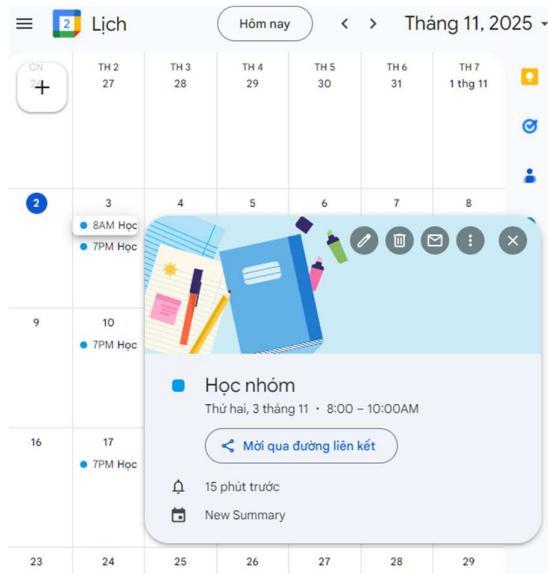
Hình 3.52 Lịch hiển thị đầy đủ được tạo bằng Neura

Tiếp theo ta tiến hành lấy sự kiện trong một khoảng thời gian ví dụ như “Từ 13h đến 16h ngày mai tôi có những lịch gì”, “Ngày mai tôi có những việc gì”,... Đồng thời ta cũng có thể lấy được thời gian rảnh của mình ví dụ “Ngày mai tôi rảnh lúc nào”, “Tôi có thời gian trống từ 10 giờ đến 14 giờ không?”



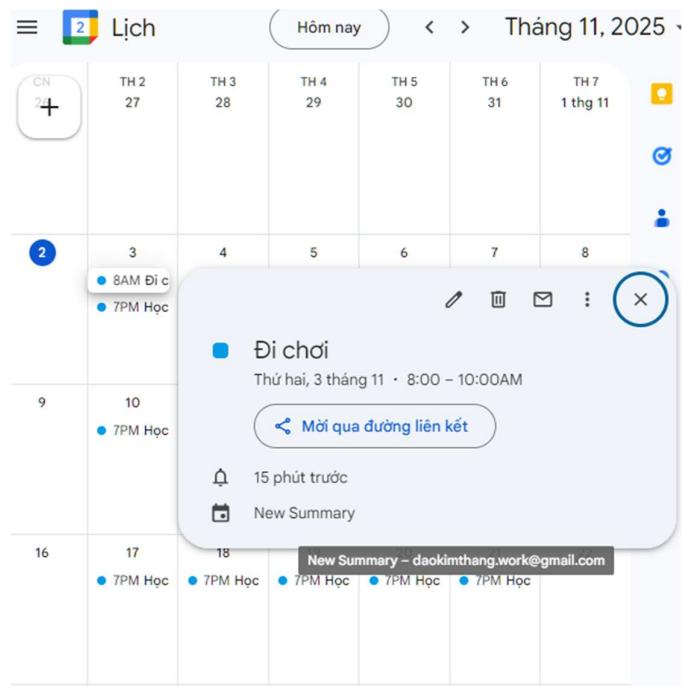
Hình 3.51 .Kết quả chạy Neura khi lấy sự kiện và thời gian rảnh

Tiếp theo ta thử thay đổi thông tin sự kiện đã có, thay đổi như tiêu đề, địa điểm hoặc thời gian của sự kiện. Ta đã tạo sự kiện “tạo lịch học nhóm từ 7h đến 9h sáng mai tại trường nhắc tôi trước 15p” thử yêu cầu chatbot đổi thời gian “đổi lịch học nhóm từ 7h đến 9h sáng mai sang 8h đến 10h sáng mai”.



Hình 3.53. Kết quả đổi thời gian sự kiện thành công

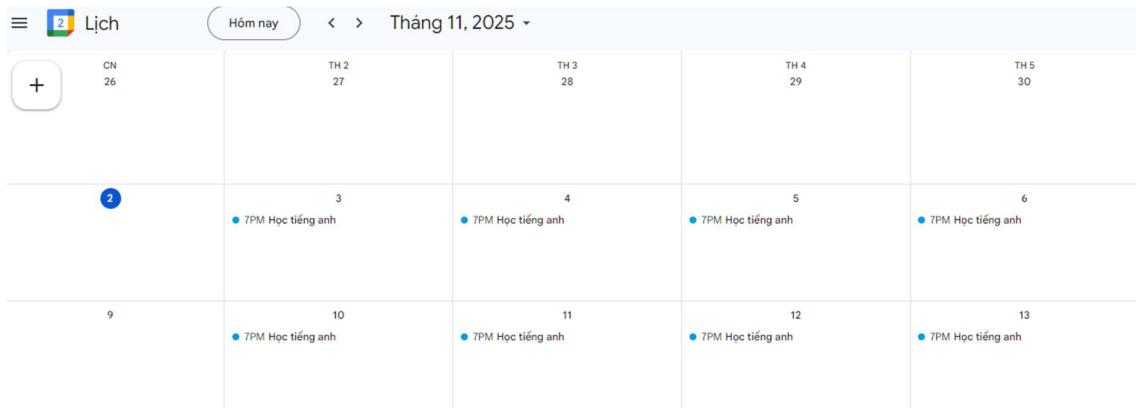
Sau đó thử đổi tiêu đề sự kiện “đổi lịch học nhóm từ 8h đến 10h sáng mai thành đi chơi vào lúc 8h đến 10h sáng mai”



Hình 3.54. Kết quả đổi tiêu đề sự kiện thành công

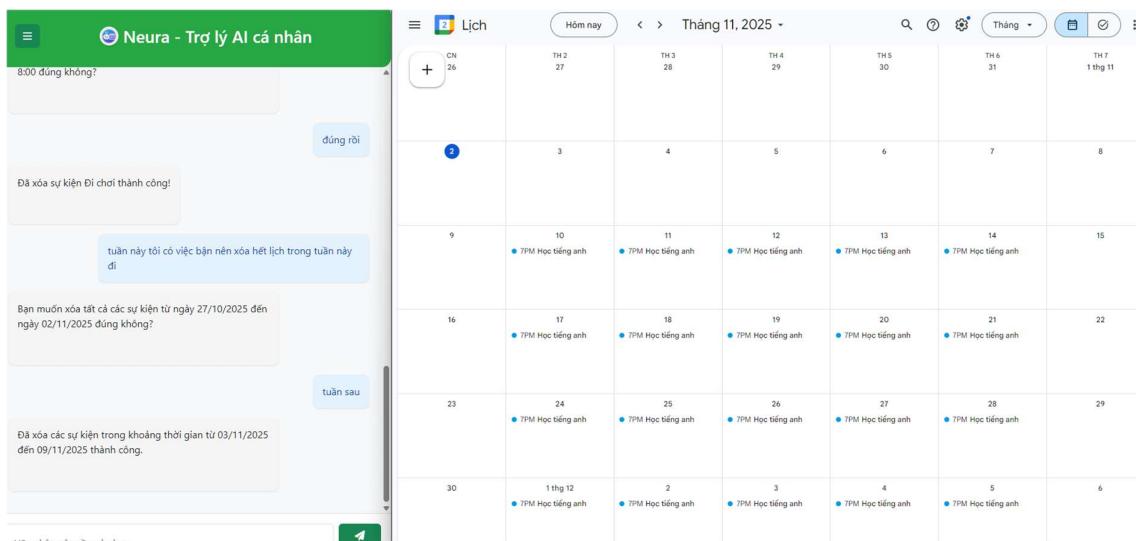
Đồ án tốt nghiệp

Tiếp theo ta xóa sự kiện đã tạo đó đi “xóa lịch đi chơi lúc 8h ngày mai đi”. Kết quả chatbot đã xóa thành công



Hình 3.55. Kết quả xóa sự kiện thành công

Ngoài xóa một sự kiện ta cũng có thể xóa nhiều sự kiện trong một khoảng thời gian ví dụ “tuần này tôi có việc bận nên xóa hết lịch trong tuần này đi”. Kết quả chatbot đã xóa được hết các lịch trong tuần

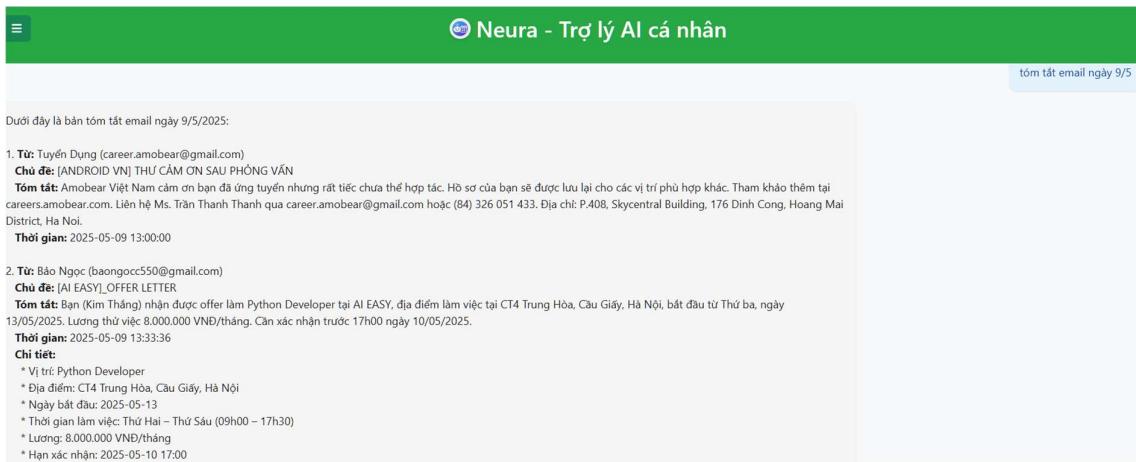


Hình 3.56. Kết quả xóa nhiều sự kiện thành công

Dựa trên việc thử nghiệm, có thể thấy chatbot có khả năng tạo, đọc, sửa và xóa (CRUD) các sự kiện trong lịch hoạt động ổn định và chính xác. Các thao tác được xử lý đúng, phản hồi từ hệ thống rõ ràng. Ngoài ra, chatbot còn có thể nhận diện thời gian, ngày tháng, tiêu đề và nội dung sự kiện từ ngôn ngữ tự nhiên, giúp người dùng thao tác dễ dàng mà không cần nhập liệu theo định dạng cố định. Việc đồng bộ dữ liệu với Google Calendar diễn ra trơn tru, đảm bảo tính nhất quán giữa các thay đổi được thực hiện qua chatbot và trên nền tảng Google. Nhìn chung, kết quả thử nghiệm cho thấy tính năng quản lý lịch của chatbot đã đạt mức hoàn thiện tốt.

3.8.2. Tác vụ liên quan đến mail

Tác vụ liên quan đến mail sẽ có những chức năng chính như tóm tắt email với khoảng thời gian cho trước, tìm thông tin, nội dung chứa trong email. Ta thử tóm tắt email vào một ngày cụ thể như “tóm tắt email ngày 9/5”



Hình 3.57. Kết quả chạy Neura khi tóm tắt email vào một thời gian cụ thể

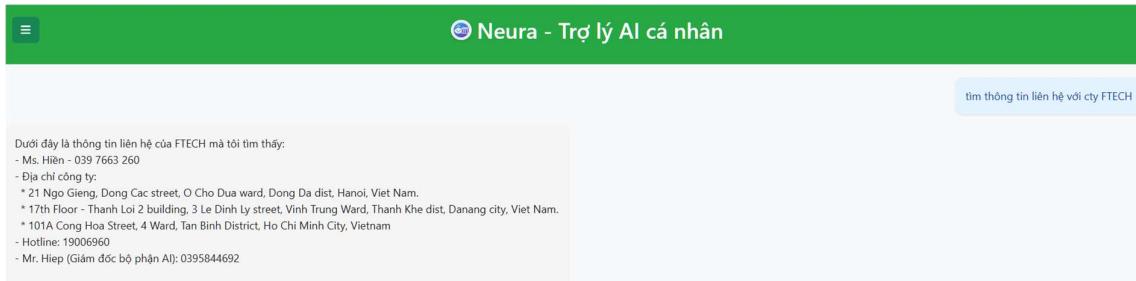
Chatbot hiểu được yêu cầu và đưa ra được bản tóm tắt của ngày 9/5, bản tóm tắt đã bao gồm được hết những thông tin quan trọng của các email. Một ví dụ khác “tóm tắt email từ ngày 10/9 đến 12/9”. Chatbot cũng đã lấy được email và đưa ra bản tóm tắt cho người dùng.



Hình 3.58. Kết quả chạy Neura khi tóm tắt email vào một khoảng thời gian

Đồ án tốt nghiệp

Tiếp theo là khả năng tìm và lấy thông tin từ các email như “tìm thông tin liên hệ với cty Ftech”. Chatbot đã dựa vào tri thức của mình đã tìm ra được thông tin liên hệ với công ty Ftech.



Hình 3.59. Kết quả chạy Neura khi tìm và truy xuất thông tin từ nội dung email

Một số ví dụ khác như “từ ngày 6/9 đến ngày 12/9 có mail phỏng vấn nào không?”. Tuy từ 6/9 đến 12/9 có nhiều mail khác nhưng không liên quan đến phỏng vấn, chatbot đã loại bỏ và không đưa ra cho người dùng mà chỉ đưa ra thông tin đúng như người dùng hỏi.



Hình 3.60. Kết quả chạy Neura khi tìm bản tóm tắt email về một nội dung cụ thể

<input type="checkbox"/>		UNIDAYS	Hỗ trợ xác minh - UNIDAYS Đề hoàn tất quy trình xác minh, hãy làm theo hướng dẫn trong email mà chúng tôi đã ...	13 Sept
<input type="checkbox"/>		info@knmholdings.vn	K&M - Thư mời phỏng vấn - Thân gửi ứng viên Kim Thắng, Bộ phận Nhân sự K&M Holdings xin chân thành cảm ơn...	12 Sept
<input type="checkbox"/>		IIG - REGISTRATION	[IIG Việt Nam] Thông báo lịch thi bài thi TOEIC LISTENING & READING - TRÊN MÁY TÍNH - Thân gửi thí sinh: Đào ...	9 Sept
<input type="checkbox"/>		Holdings KnM	Đã đặt cuộc hẹn: K&M Job Interview Booking (Thắng Đào Kim) @ th 2, 8 Tháng 9, 2025 10AM - 11AM (GMT+7) (d...)	6 Sept
<input type="checkbox"/>		info@knmhol., me 2	K&M - Job Interview Invitation - Kính gửi Quý công ty, Tôi xin chân thành cảm ơn công ty đã mời tôi tham dự buổi...	6 Sept
<input type="checkbox"/>		Anthropic Team	We're Updating our Consumer Terms and Privacy Policy - Claude Hello, We're writing to inform you about import...	5 Sept

Hình 3.61. Danh sách email từ 6/9 đến 12/9

Những trường hợp trên cho thấy chatbot xử lý tốt các tác vụ liên quan đến mail, đáp ứng được các yêu cầu, chức năng đề ra. Điều này chứng tỏ hệ thống không chỉ dừng lại ở việc đọc hiểu nội dung email mà còn có khả năng phân tích ngữ nghĩa, nhận diện chính xác mục đích truy vấn của người dùng. Chatbot có thể lọc và xử lý thông tin theo ngữ cảnh, giúp kết quả trả về mang tính chọn lọc, phù hợp và sát với nhu cầu thực tế. Bên cạnh đó, khả năng tương tác tự nhiên của chatbot cũng được thể hiện rõ ràng. Người dùng có thể sử dụng ngôn ngữ linh hoạt, không cần tuân theo cú pháp cứng nhắc, mà hệ thống vẫn hiểu đúng và phản hồi chính xác. Điều này giúp trải nghiệm người dùng trở nên thân thiện và gần gũi hơn.

Đồ án tốt nghiệp

Ngoài hai chức năng chính là tóm tắt email và tìm kiếm thông tin trong email, hệ thống còn có tiềm năng mở rộng thêm các tính năng nâng cao như:

- Tự động phân loại email theo chủ đề (công việc, cá nhân, thông báo, deadline, v.v.).
- Phát hiện thời gian và sự kiện trong nội dung thư để gợi ý tạo lịch trên Google Calendar (đã có thể nhưng chưa ổn định).
- Tổng hợp báo cáo định kỳ về số lượng email nhận được, tỷ lệ phản hồi, hoặc các chủ đề thường xuất hiện.

Từ đó, có thể khẳng định rằng chatbot đã đáp ứng tốt các yêu cầu đặt ra về mặt chức năng, khả năng xử lý ngôn ngữ tự nhiên, và độ chính xác trong việc truy xuất thông tin. Hệ thống hoạt động ổn định, thông minh, và có tính ứng dụng cao trong thực tế đặc biệt trong việc hỗ trợ người dùng quản lý email hiệu quả hơn.

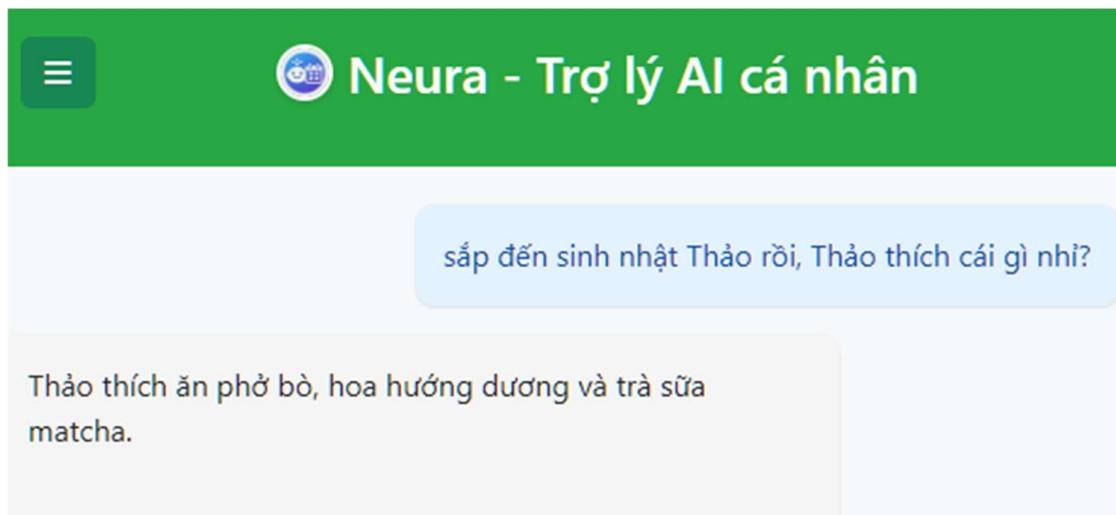
3.8.3. Tác vụ khác

Ngoài các tác vụ liên quan đến lịch và email, chatbot có thể ghi nhớ và truy xuất thông tin đã ghi nhớ, tạo các tác vụ tự động hàng ngày, tuần, rồi gửi kết quả tới Bot(Telegram, Zalo).

Ví dụ ta có thể cung cấp thông tin cho chatbot ghi nhớ, sau khi cung cấp chatbot sẽ tiến hành ghi nhớ vào cơ sở dữ liệu khi cần người dùng chỉ cần hỏi đến chatbot có thể đưa ra được thông tin.

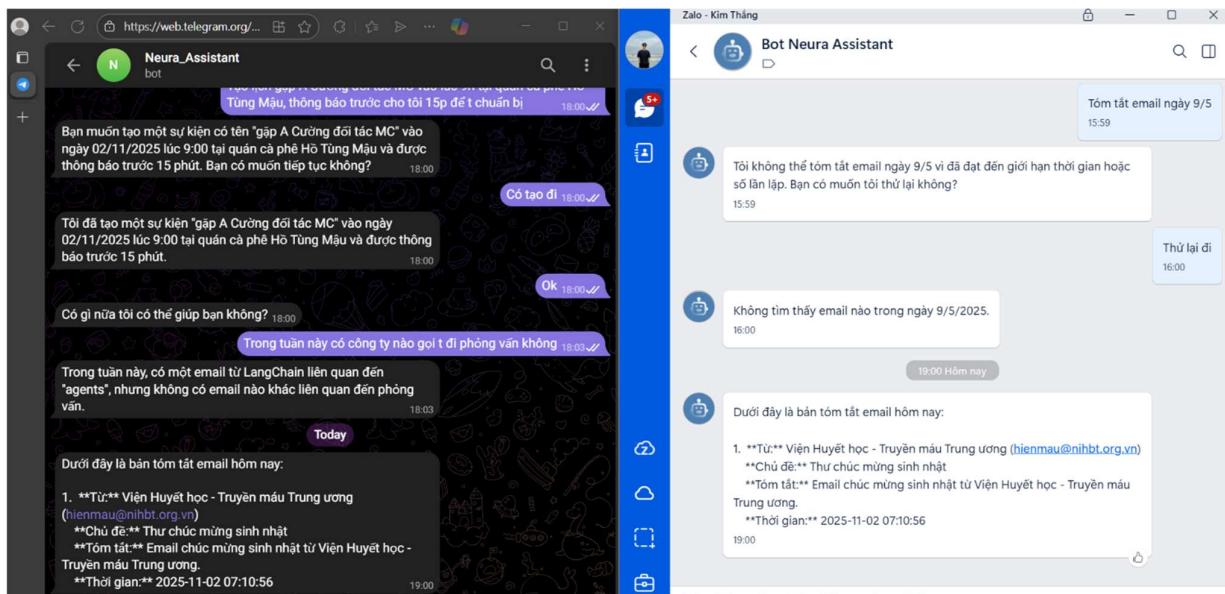


Hình 3.62. Cung cấp thông tin cần ghi nhớ cho chatbot



Hình 3.63. Thông tin chatbot đưa ra khi được hỏi đến dữ liệu đã lưu

Ngoài ra ta còn có thể tạo một nhiệm vụ tự động để chatbot có thể làm ví dụ như “Tạo lịch tóm tắt email hàng ngày tự động vào lúc 18h40”. Chatbot sẽ tiến hành trích xuất thời gian và yêu cầu nhiệm vụ sau đó sẽ lưu lại. Hàng ngày cứ đến 18h40 hệ thống sẽ tiến hành gọi hàm thực thi agent manager sau đó tiến hành thực hiện nhiệm vụ, kết quả sau đó sẽ được thông báo về bot(telegram, zalo).



Hình 3.64. Kết quả nhiệm vụ tự động được gửi về bot Telegram và Zalo

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Đồ án “Xây dựng trợ lý AI cá nhân – Neura” được thực hiện với mục tiêu phát triển một hệ thống trợ lý ảo có khả năng tương tác tự nhiên bằng ngôn ngữ con người, hỗ trợ người dùng trong việc quản lý lịch làm việc, xử lý email và ghi nhớ thông tin cá nhân. Sau quá trình nghiên cứu và triển khai, hệ thống đã đạt được nhiều kết quả tích cực.

Cụ thể, Neura đã được xây dựng với khả năng hiểu và phản hồi ngôn ngữ tự nhiên, sử dụng mô hình ngôn ngữ lớn (LLM) để xử lý hội thoại, giúp việc giao tiếp giữa người dùng và hệ thống trở nên linh hoạt và tự nhiên hơn. Hệ thống đã tích hợp thành công Google Calendar API, cho phép thực hiện đầy đủ các thao tác như tạo, đọc, cập nhật và xóa sự kiện, đồng thời có thể truy xuất thông tin sự kiện, kiểm tra thời gian rảnh và tạo lịch trực tiếp bằng ngôn ngữ tự nhiên. Bên cạnh đó, Neura còn sử dụng Google Gmail API để lấy và xử lý email. Thông qua LLM, hệ thống có thể tóm tắt nội dung email, giúp người dùng nhanh chóng nắm bắt thông tin quan trọng, đồng thời cho phép lưu trữ và truy xuất email bằng ngôn ngữ tự nhiên, giúp việc tìm kiếm trở nên thuận tiện hơn. Ngoài ra, kỹ thuật RAG (Retrieval-Augmented Generation) cũng được áp dụng để giúp hệ thống truy xuất và sử dụng lại các thông tin đã được lưu trữ hoặc lấy từ email, góp phần giảm thiểu hiện tượng “ảo giác” của mô hình ngôn ngữ và nâng cao độ chính xác trong phản hồi. Một điểm nổi bật khác của Neura là khả năng tự động hóa nhiệm vụ. Hệ thống có thể hiểu lệnh của người dùng như “Tạo lịch tóm tắt email hàng ngày tự động lúc 19h00”, sau đó tự động trích xuất thời gian, lưu nhiệm vụ và đến đúng thời điểm sẽ thực thi tác vụ, gửi kết quả tóm tắt email về bot Telegram hoặc Zalo. Nhờ đó, Neura không chỉ phản hồi theo yêu cầu mà còn có thể hoạt động chủ động, hỗ trợ người dùng một cách tự động và liên tục. Cuối cùng, hệ thống đã được triển khai đa nền tảng, bao gồm giao diện web (xây dựng bằng Flask), bot Telegram và bot Zalo, tất cả đều vận hành ổn định dựa trên cơ chế polling.

Kết quả đạt được cho thấy việc kết hợp giữa mô hình ngôn ngữ lớn (LLM), kỹ thuật RAG và các API thực tế hoàn toàn khả thi trong việc tạo nên một trợ lý ảo có tính ứng dụng cao. Neura không chỉ dừng lại ở việc trả lời câu hỏi, mà còn có khả năng hỗ trợ quản lý công việc cá nhân một cách thông minh và tự động hơn. Quá trình triển khai cũng giúp người thực hiện củng cố kiến thức về lập trình, xử lý ngôn ngữ tự nhiên, tích hợp API và phát triển ứng dụng thực tế.

Tuy nhiên chatbot vẫn còn một số hạn chế nhất định. Khả năng ghi nhớ và cá nhân hóa của Neura còn ở mức cơ bản, các thông tin khi truy xuất vẫn thừa thêm các thông tin khác khi truy xuất. Ngoài ra thời gian xử lý của chatbot phản hồi lâu vì phải qua xử lý của Multi-agent đó cũng là một số giới hạn trong việc sử dụng thư viện Langchain. Mặc dù LangChain hỗ trợ tốt cho việc kết nối mô hình ngôn ngữ lớn (LLM) với các công cụ và nguồn dữ liệu bên ngoài, nhưng khi hệ thống ngày càng phức tạp, việc quản lý luồng xử lý, trạng thái hội thoại và các nhánh logic trở nên khó

kiểm soát. Vì vậy, hướng phát triển tiếp theo của dự án là chuyển từ Langchain sang LangGraph – một kiến trúc mới dựa trên đồ thị luồng (graph-based workflow). LangGraph cho phép định nghĩa rõ ràng các node, edge và trạng thái hội thoại, giúp hệ thống trở nên trực quan, dễ mở rộng và dễ kiểm soát hơn. Điều này không chỉ giúp tăng tính ổn định và khả năng mở rộng, mà còn phù hợp với định hướng phát triển Neura trở thành một trợ lý AI đa tác vụ, hoạt động theo mô hình agent hợp tác. Khi đó việc tích hợp thêm các dịch vụ khác như Google Workspace trở nên dễ dàng hơn, đồng thời áp dụng các kỹ thuật RAG nâng cao để tăng độ chính xác khi truy xuất thông tin hơn.

TÀI LIỆU THAM KHẢO

- [1] “Doanh nghiệp công nghệ số hãy nhận lấy sứ mệnh dùng chuyển đổi AI để thay đổi Việt Nam”, <https://mst.gov.vn/doanh-nghiep-cong-nghe-so-hay-nhan-lay-su-men-dung-chuyen-doi-ai-de-thay-doi-viet-nam-197240915191157566.htm>
- [2] J. Singh, *Natural Language Processing in the Real World*, CRC Press, 2023.
- [3] “LLM (Mô hình ngôn ngữ lớn) là gì?”, <https://aws.amazon.com/vi/what-is/large-language-model/>
- [4] Job Scheduling in Python with APScheduler | Better Stack Community, <https://betterstack.com/community/guides/scaling-python/apscheduler-scheduled-tasks/>, [17/08]
- [5] “Day 1: Introduction to Retrieval Augmented Generation“, <https://bit.ly/4lgxkkk>
- [6] “Exploring LangChain”, <https://bit.ly/4lgvWy4>
- [7] ‘Flask – (Creating first simple application)’, <https://www.geeksforgeeks.org/flask-creating-first-simple-application/>
- [8] Hướng dẫn xây dựng Zalo Bot đơn giản với cơ chế Polling, <https://bot.zapps.me/docs/build-your-bot/>