✅ Day3

Today's Assignments

Complete Unit 3: "Generative AI Agents", which is:

- [Optional] Listen to the summary podcast episode for this unit (created by NotebookLM).
- Read the "Generative AI Agents" whitepaper.
- Complete these code labs on Kaggle:
  1. Talk to a database with function calling
  2. Build an agentic ordering system in LangGraph

What You'll Learn

Learn to build sophisticated AI agents by understanding their core components and the iterative development process.

The code labs cover how to connect LLMs to existing systems and to the real world. Learn about function calling by giving SQL tools to a chatbot, and learn how to build a LangGraph agent that takes orders in a café.

Reminders and Announcements

- Here is the recording from Day 2's livestream.
- The next livestream is tomorrow at 2pm PST/ 5pm EST/ 10pm UTC. Click here to join! Livestream guests: Alan Blount, Wes Dyer, Steven Johnson, Patrick Marlow, Anant Nawalgaria, and Julia Wiesinger.
- Find a complete list of scheduled livestreams and past recordings here.
- Be sure to ask all your questions about the podcast, readings, and code lab on Discord.
- Additionally, we've added 2 new channels on Discord to enhance discussion:
    - #5dgai-question-forum is a Discord forum (a special type of channel) where you can create specific threads, which will help finding answers easier in the future.
    - #5dgai-course-content is a Discord channel for deeper discussion of course content only (excluding technical troubleshooting questions).

Happy learning and see you tomorrow,

The Kaggle Team

Agents

Tác giả : Julia Wiesinger, Patrick Marlow and Vladimir Vuskovic

Giới thiệu:

Con người rất giỏi trong việc nhận diện các mẫu phức tạp, không rõ ràng. Tuy nhiên, họ thường phải dựa vào các công cụ hỗ trợ – như sách, Google Tìm kiếm,

hoặc máy tính cầm tay – để bổ sung kiến thức trước khi đưa ra kết luận cuối cùng. Tương tự như vậy, các mô hình `Generative AI` (Trí tuệ nhân tạo tạo sinh) cũng có thể được huấn luyện để sử dụng các công cụ nhằm truy cập thông tin theo thời gian thực hoặc đề xuất các hành động thực tế. Ví dụ, một mô hình có thể tận dụng công cụ truy vấn cơ sở dữ liệu để truy cập thông tin cụ thể, chẳng hạn như lịch sử mua hàng của khách hàng, giúp tạo ra các đề xuất mua sắm cá nhân hóa. Ngoài ra, dựa trên truy vấn của người dùng, mô hình có thể thực hiện các lệnh gọi API để gửi phản hồi email cho đồng nghiệp hoặc hoàn tất giao dịch tài chính thay mặt người dùng. Để làm được điều này, mô hình không chỉ cần quyền truy cập vào một tập hợp các công cụ bên ngoài, mà còn phải có khả năng lập kế hoạch và thực thi các nhiệm vụ theo cách tự định hướng. Sự kết hợp giữa khả năng suy luận, logic và truy cập thông tin ngoại vi, tất cả kết nối với một mô hình Generative AI, gợi lên khái niệm về `"agent"` (tác nhân), hay một chương trình mở rộng vượt xa khả năng độc lập của một mô hình Generative AI. Bài báo này sẽ đi sâu vào các khía cạnh này và các vấn đề liên quan một cách chi tiết hơn.

Giới thiệu:

What is an agent? (Tác nhân là gì?)

Ở dạng cơ bản nhất, một **Generative AI agent (tác nhân AI tạo sinh)** có thể được định nghĩa là một ứng dụng cố gắng đạt được mục tiêu bằng cách quan sát thế giới và thực hiện các hành động dựa trên các công cụ mà nó có sẵn. Các agent này hoạt động **tự động** và có thể hành động độc lập mà không cần sự can thiệp của con người, đặc biệt là khi được cung cấp các mục tiêu hoặc nhiệm vụ rõ ràng mà chúng cần hoàn thành. Các agent cũng có thể chủ động tiếp cận mục tiêu của mình. Ngay cả khi không có các hướng dẫn cụ thể từ con người, một agent có thể suy luận về bước tiếp theo cần thực hiện để đạt được mục tiêu cuối cùng. Mặc dù khái niệm về agent trong AI rất rộng và mạnh mẽ, whitepaper này tập trung vào các loại agent cụ thể mà các mô hình Generative AI có thể xây dựng tại thời điểm công bố.

Để hiểu được cách hoạt động bên trong của một agent, trước tiên chúng ta cần giới thiệu các thành phần cơ bản điều khiển hành vi, hành động và quá trình ra quyết định của agent (behavior, actions, and decision making). Sự kết hợp của các thành phần này có thể được mô tả như là một **kiến trúc nhận thức (cognitive architecture)**, và có rất nhiều kiến trúc như vậy được tạo ra bằng cách kết hợp các thành phần này với nhau. Tập trung vào các chức năng cốt lõi, có **ba thành phần thiết yếu** trong kiến trúc nhận thức của một agent, được minh họa trong Hình 1.

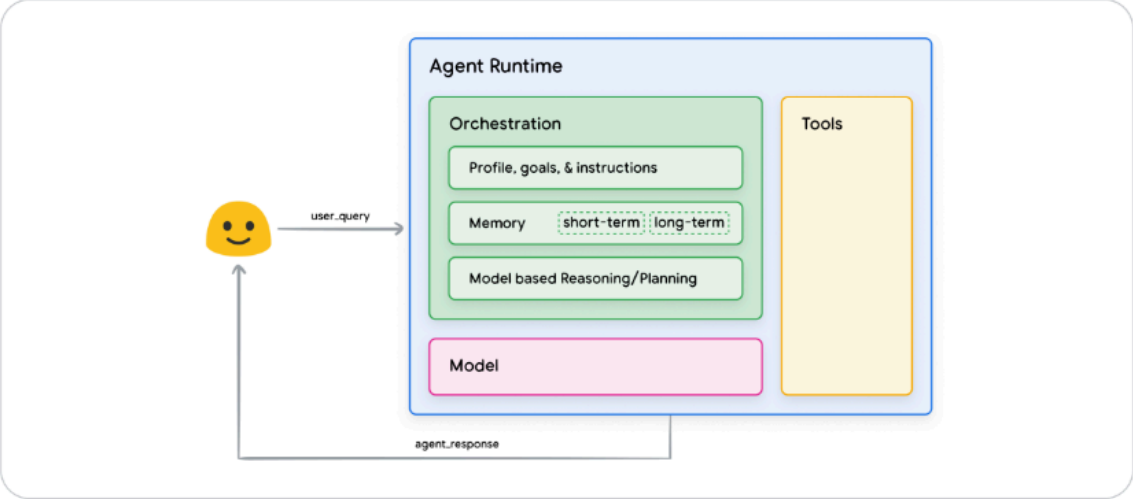


Figure 1: Kiến trúc tổng quát của tác nhân và các thành phần

The model (Mô hình)

Trong phạm vi của một agent, **mô hình** đề cập đến mô hình ngôn ngữ (**Language Model - LM**) được sử dụng như là **bộ xử lý quyết định trung tâm** cho các quy trình của agent. Mô hình mà agent sử dụng có thể là một hoặc nhiều mô hình ngôn ngữ, có kích thước bất kỳ (nhỏ hoặc lớn), có khả năng tuân theo các khung logic và suy luận dựa trên hướng dẫn, như **ReAct**, **Chain-of-Thought**, hoặc **Tree-of-Thoughts**.

Các mô hình có thể là đa mục đích (general purpose), đa phương thức (multimodal) hoặc đã được tinh chỉnh (**fine-tuned**) dựa trên nhu cầu của kiến trúc agent cụ thể. Để đạt kết quả sản xuất tốt nhất, bạn nên tận dụng một mô hình phù hợp nhất với ứng dụng cuối cùng mong muốn, và lý tưởng nhất là mô hình này đã được huấn luyện trên các dữ liệu liên quan đến các công cụ mà bạn dự định sử dụng trong kiến trúc nhận thức.

Điều quan trọng cần lưu ý là mô hình thường không được huấn luyện với các cấu hình cụ thể (ví dụ: lựa chọn công cụ, điều phối, thiết lập suy luận- tool choices, orchestration/reasoning setup) của agent. Tuy nhiên, có thể tinh chỉnh thêm mô hình cho các nhiệm vụ của agent bằng cách cung cấp cho nó các ví dụ thể hiện khả năng của agent, bao gồm các trường hợp agent sử dụng công cụ cụ thể hoặc các bước suy luận trong nhiều ngữ cảnh khác nhau.

The tools (Các công cụ)

Các mô hình nền tảng (**foundational models**), mặc dù có khả năng ấn tượng trong việc tạo ra văn bản và hình ảnh, vẫn bị giới hạn bởi việc không thể tương tác với thế giới bên ngoài. **Tool** - **Công cụ** chính là yếu tố kết nối, giúp các agent có khả năng tương tác với dữ liệu và dịch vụ bên ngoài, mở rộng phạm vi hành động vượt xa khả năng của mô hình ban đầu. Công cụ có thể có nhiều hình thức khác nhau và có độ phức tạp khác nhau, nhưng thường tuân theo các phương thức API web phổ biến như **GET**, **POST**, **PATCH**, và **DELETE**.

Ví dụ, một công cụ có thể cập nhật thông tin khách hàng trong cơ sở dữ liệu hoặc lấy dữ liệu thời tiết để đưa ra đề xuất du lịch mà agent đang cung cấp cho người dùng. Với sự hỗ trợ của các công cụ, các agent có thể truy cập và xử lý thông tin từ thế giới thực. Điều này cho phép chúng hỗ trợ các hệ thống chuyên biệt hơn, như R**etrieval Augmented Generation (RAG)**, giúp mở rộng đáng kể khả năng của agent, vượt xa những gì mà mô hình nền tảng có thể đạt được một cách độc lập.

Chúng tôi sẽ thảo luận chi tiết hơn về các công cụ ở phần dưới, nhưng điều quan trọng nhất cần hiểu là các công cụ chính là cầu nối giữa khả năng nội tại của agent và thế giới bên ngoài, mở ra nhiều tiềm năng mới.

The orchestration layer (Lớp điều phối)

Lớp orchestration (điều phối) mô tả một **quá trình lặp đi lặp lại** nhằm quản lý cách mà agent tiếp nhận thông tin, thực hiện một số suy luận nội bộ, và sử dụng suy luận đó để quyết định hành động hoặc quyết định tiếp theo. Thông thường, quá trình lặp này sẽ tiếp diễn cho đến khi agent đạt được mục tiêu của mình hoặc gặp phải một điểm dừng.

Độ phức tạp của lớp điều phối có thể thay đổi đáng kể tùy thuộc vào agent và nhiệm vụ mà nó đang thực hiện. Một số vòng lặp có thể chỉ là các tính toán đơn giản với các quy tắc quyết định, trong khi các vòng lặp khác có thể chứa các logic liên kết, bao gồm các thuật toán máy học bổ sung, hoặc áp dụng các kỹ thuật suy luận xác suất khác.

Chúng tôi sẽ thảo luận chi tiết hơn về việc triển khai lớp điều phối của agent trong phần kiến trúc nhận thức.

Agents vs models

Để hiểu rõ hơn về sự khác biệt giữa
agent (tác tử) và **model** (mô hình), hãy xem xét biểu đồ sau:

Models (Mô hình)	**Agents (Tác nhân)**
Kiến thức bị giới hạn trong dữ liệu huấn luyện có sẵn.	Kiến thức được mở rộng thông qua kết nối với các hệ thống bên ngoài thông qua các công cụ.
Chỉ thực hiện suy luận hoặc dự đoán một lần dựa trên truy vấn của người dùng. Nếu không được triển khai rõ ràng, mô hình sẽ không quản lý lịch sử phiên hoặc ngữ cảnh liên tục (ví dụ: lịch sử chat).	Quản lý lịch sử phiên (ví dụ: lịch sử chat) cho phép suy luận/dự đoán đa vòng dựa trên các truy vấn của người dùng và các quyết định được thực hiện trong lớp điều phối. Trong ngữ cảnh này, một **“turn”** được định nghĩa là một tương tác giữa hệ thống và agent (ví dụ: 1 sự kiện/truy vấn đầu vào và 1 phản hồi của agent).
Không có triển khai công cụ nội tại.	Các công cụ được triển khai tích hợp trong kiến trúc của agent.
Không có lớp logic nội tại. Người dùng có thể tạo các câu hỏi đơn giản hoặc sử dụng các khung suy luận như CoT, ReAct để hình thành các câu hỏi phức tạp nhằm hướng dẫn mô hình trong quá trình dự đoán.	Có kiến trúc nhận thức nội tại sử dụng các khung suy luận như Chain-of-Thought (CoT), ReAct, hoặc các khung agent có sẵn như LangChain.

Cognitive Architectures (Cách hoạt động của các agent)

- Hình dung một đầu bếp làm việc trong một nhà bếp bận rộn. Mục tiêu của họ là tạo ra các món ăn ngon cho khách hàng của nhà hàng, liên quan đến một chu trình gồm lập kế hoạch, thực hiện và điều chỉnh.
- **Quá trình hoạt động:**
    - Họ thu thập thông tin, chẳng hạn như đơn đặt món của khách và các nguyên liệu hiện có trong kho và tủ lạnh.
    - Họ thực hiện suy luận nội bộ về các món ăn và hương vị có thể tạo ra dựa trên thông tin đã thu thập.
    - Họ tiến hành thực hiện món ăn: cắt rau, pha trộn gia vị, nướng thịt.
- Ở mỗi giai đoạn, đầu bếp thực hiện điều chỉnh khi cần thiết, tinh chỉnh kế hoạch nếu nguyên liệu thiếu hụt hoặc khi nhận được phản hồi từ khách hàng. Chu trình này gồm thu thập thông tin, lập kế hoạch, thực hiện và điều chỉnh mô tả một kiến trúc nhận thức độc đáo mà đầu bếp sử dụng để đạt được mục tiêu.

- Giống như đầu bếp, các agent có thể sử dụng kiến trúc nhận thức để đạt được mục tiêu cuối cùng của mình thông qua quy trình xử lý thông tin lặp đi lặp lại, đưa ra quyết định có căn cứ và điều chỉnh hành động tiếp theo dựa trên kết quả trước đó. Trọng tâm của kiến trúc nhận thức dành cho các agent là lớp điều phối (**orchestration layer)**, lớp này chịu trách nhiệm duy trì bộ nhớ, trạng thái, suy luận và lập kế hoạch. Nó sử dụng các phương pháp hiện đại trong lĩnh vực **prompt engineering** và các framework liên quan để hướng dẫn quá trình suy luận và lập kế hoạch, giúp agent tương tác hiệu quả hơn với môi trường và hoàn thành nhiệm vụ. Nghiên cứu trong lĩnh vực prompt engineering và lập kế hoạch nhiệm vụ cho các mô hình ngôn ngữ đang phát triển nhanh chóng, mang lại nhiều phương pháp đầy hứa hẹn. Mặc dù không phải là danh sách đầy đủ, dưới đây là một số framework và kỹ thuật suy luận phổ biến nhất tại thời điểm bài viết này:
    - **ReAct**: Một framework về prompt engineering cung cấp chiến lược quá trình suy luận cho các mô hình ngôn ngữ nhằm **Reason (Suy luận)** và **Act (Hành động)** dựa trên truy vấn của người dùng, có hoặc không kèm theo các ví dụ trong ngữ cảnh. ReAct prompting đã cho thấy hiệu suất vượt trội

so với một số baseline SOTA (State-of-the-Art) và cải thiện khả năng tương tác giữa người và mô hình cũng như độ tin cậy của các mô hình ngôn ngữ lớn (LLMs).

- **Chain-of-Thought (CoT)**: Một framework về prompt engineering giúp tăng cường khả năng suy luận thông qua các bước trung gian. Có nhiều kỹ thuật phụ trong CoT bao gồm:
  - **Self-consistency (Tự nhất quán)**
  - **Active-prompt (Nhắc chủ động)**
  - **Multimodal CoT (CoT đa phương thức)**

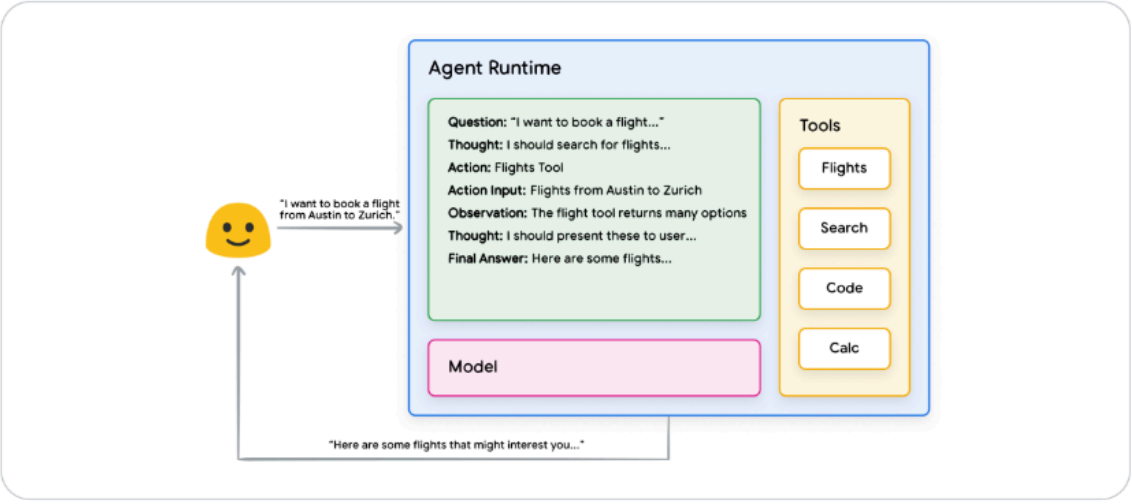
  Mỗi kỹ thuật này đều có điểm mạnh và điểm yếu tùy thuộc vào ứng dụng cụ thể.
- **Tree-of-Thoughts (ToT)**: Một framework về prompt engineering phù hợp cho các tác vụ khám phá hoặc chiến lược nhìn xa (strategic lookahead). ToT tổng quát hóa từ CoT, cho phép mô hình khám phá nhiều chuỗi suy nghĩ khác nhau như các bước trung gian để giải quyết vấn đề chung với các mô hình ngôn ngữ.

Các agent có thể sử dụng một trong các kỹ thuật suy luận ở trên, hoặc nhiều kỹ thuật khác, để chọn hành động tiếp theo tốt nhất cho yêu cầu của người dùng. Ví dụ, hãy xem xét một agent được lập trình để sử dụng framework **ReAct** nhằm chọn hành động và công cụ phù hợp cho truy vấn của người dùng. Trình tự các bước có thể diễn ra như sau:

1. **Người dùng gửi truy vấn đến agent**.
2. **Agent bắt đầu chuỗi ReAct**.
3. **Agent cung cấp một prompt cho mô hình**, yêu cầu mô hình tạo ra một trong các bước tiếp theo của ReAct và đầu ra tương ứng:
   - **a. Question (Câu hỏi)**: Đây là câu hỏi đầu vào từ truy vấn của người dùng, được cung cấp qua prompt.
   - **b. Thought (Suy nghĩ)**: Suy nghĩ của mô hình về những gì cần làm tiếp theo.

- **c. Action (Hành động)**: Quyết định của mô hình về hành động cần thực hiện tiếp theo.
    - Đây là bước mà mô hình sẽ lựa chọn công cụ.
    - Ví dụ, một hành động có thể là một trong các lựa chọn: **[Flights, Search, Code, None]**, trong đó 3 lựa chọn đầu tiên đại diện cho các công cụ mà mô hình có thể sử dụng, và lựa chọn cuối cùng ("None") đại diện cho "không chọn công cụ".
- **d. Action input (Đầu vào cho hành động)**: Quyết định của mô hình về thông tin đầu vào cần cung cấp cho công cụ (tool) (nếu có).
- **e. Observation (Quan sát)**: Kết quả của chuỗi hành động (action) và thông tin đầu vào (action input sequence).
    - Chu trình **Thought / Action / Action input / Observation** có thể lặp lại **N** lần khi cần thiết.
- **f. Final answer (Câu trả lời cuối cùng)**: Câu trả lời cuối cùng của mô hình để gửi lại cho truy vấn của người dùng.

4. **Chu trình ReAct kết thúc và câu trả lời cuối cùng được gửi lại cho người dùng**.



Hình 2: Ví dụ về agent sử dụng suy luận ReAct trong lớp điều phối (orchestration layer)

Như đã chỉ ra trong Hình 2, mô hình, các công cụ và cấu hình của agent làm việc

cùng nhau để cung cấp một phản hồi rõ ràng và có cơ sở cho người dùng dựa trên truy vấn gốc của họ. Trong khi mô hình có thể đoán câu trả lời dựa trên kiến thức đã có (có thể gây ra "hallucination" - sự suy diễn không chính xác), thì nó sẽ sử dụng công cụ (ví dụ như
Flights) để tìm kiếm thông tin bên ngoài theo thời gian thực. Thông tin bổ sung này được cung cấp cho mô hình, cho phép nó đưa ra quyết định chính xác hơn dựa trên dữ liệu thực tế và tóm tắt thông tin này lại cho người dùng.

Chất lượng phản hồi của agent có thể được liên kết trực tiếp với khả năng của mô hình trong việc suy luận và thực hiện các tác vụ khác nhau, bao gồm khả năng lựa chọn đúng công cụ và cách định nghĩa tốt các công cụ đó. Giống như một đầu bếp chế biến món ăn với các nguyên liệu tươi và chú ý đến phản hồi của khách hàng, các agent dựa vào suy luận hợp lý và thông tin đáng tin cậy để đưa ra kết quả tối ưu. Trong phần tiếp theo, chúng ta sẽ khám phá các cách khác nhau mà các agent kết nối với dữ liệu mới.

Công cụ: Chìa khóa kết nối với thế giới bên ngoài

Mặc dù các mô hình ngôn ngữ rất giỏi trong việc xử lý thông tin, nhưng chúng thiếu khả năng trực tiếp nhận thức và tác động đến thế giới thực. Điều này giới hạn tính hữu ích của chúng trong các tình huống yêu cầu tương tác với các hệ thống hoặc dữ liệu bên ngoài. Điều này có nghĩa là, theo một cách nào đó, mô hình ngôn ngữ chỉ tốt như những gì nó đã học từ dữ liệu huấn luyện. Nhưng cho dù chúng ta cung cấp bao nhiêu dữ liệu cho mô hình, nó vẫn thiếu khả năng cơ bản để tương tác với thế giới bên ngoài. Vậy làm thế nào để chúng ta có thể trang bị cho các mô hình khả năng tương tác theo thời gian thực và có ý thức về ngữ cảnh với các hệ thống bên ngoài? Các chức năng (Functions), tiện ích mở rộng (Extensions), kho lưu trữ dữ liệu (Data Stores) và các plugin đều là những cách để cung cấp khả năng quan trọng này cho mô hình.

Mặc dù có nhiều tên gọi khác nhau, **các công cụ** (tools) chính là cầu nối giữa các mô hình nền tảng và thế giới bên ngoài. Sự liên kết này với các hệ thống và dữ liệu bên ngoài cho phép các agent thực hiện nhiều tác vụ hơn với độ chính xác và độ

tin cậy cao hơn. Ví dụ, các công cụ (tool) có thể cho phép agent điều chỉnh cài đặt nhà thông minh, cập nhật lịch, lấy thông tin người dùng từ cơ sở dữ liệu hoặc gửi email dựa trên một tập hợp hướng dẫn cụ thể.

Tính đến thời điểm công bố bài viết này, có ba loại công cụ chính mà các mô hình của Google có thể tương tác: **Tiện ích mở rộng (Extensions)**, **Chức năng (Functions)** và **Kho lưu trữ dữ liệu (Data Stores)**. Bằng cách trang bị cho các agent các công cụ này, chúng ta mở ra tiềm năng lớn cho chúng không chỉ hiểu thế giới mà còn có thể hành động, mở ra cánh cửa cho vô số ứng dụng và khả năng mới.

Extensions (Tiện ích mở rộng)

Cách dễ nhất để hiểu **Extensions** là nghĩ đến chúng như là cầu nối giữa một API và một agent theo một cách chuẩn hóa, cho phép các agent thực thi API một cách liền mạch bất kể cách thức triển khai bên dưới như thế nào. Ví dụ, giả sử bạn đã xây dựng một agent với mục tiêu giúp người dùng đặt vé máy bay. Bạn biết rằng mình muốn sử dụng API của Google Flights để lấy thông tin chuyến bay, nhưng bạn chưa biết làm cách nào để agent của bạn có thể thực hiện cuộc gọi đến endpoint của API này.

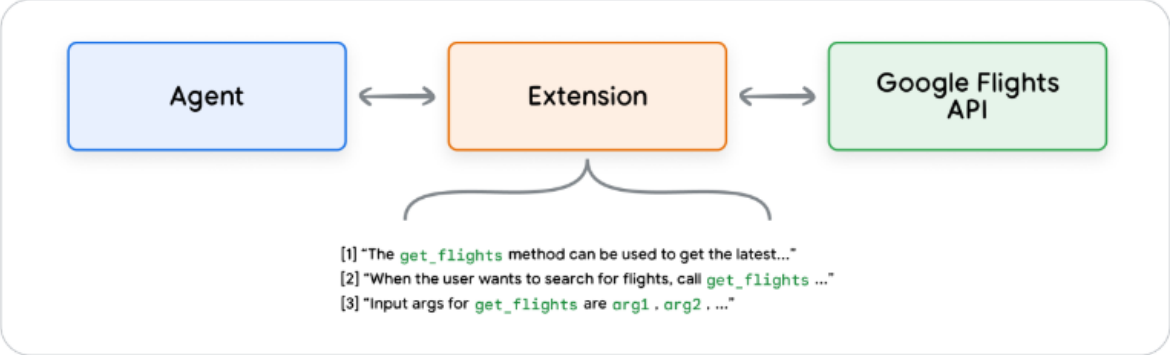
Hình 3: Làm thế nào các agent tương tác với các API bên ngoài?

Một cách tiếp cận có thể là xây dựng một đoạn mã tùy chỉnh để xử lý truy vấn người dùng, phân tích cú pháp truy vấn để lấy thông tin liên quan, sau đó thực hiện cuộc gọi API. Ví dụ, trong một trường hợp sử dụng đặt vé máy bay, người dùng có thể nói:
"Tôi muốn đặt vé máy bay từ Austin đến Zurich.". Trong kịch bản này, đoạn mã tùy chỉnh của chúng ta sẽ cần trích xuất "Austin" và "Zurich" như là các thông tin

quan trọng từ truy vấn của người dùng trước khi cố gắng thực hiện cuộc gọi API. Tuy nhiên, điều gì sẽ xảy ra nếu người dùng chỉ nói: **"Tôi muốn đặt vé máy bay đến Zurich"** và không cung cấp thành phố khởi hành? Lúc này, cuộc gọi API sẽ thất bại do thiếu dữ liệu cần thiết, và chúng ta sẽ phải viết thêm mã để xử lý các trường hợp đặc biệt này. Phương pháp này không có khả năng mở rộng và dễ gặp lỗi trong các tình huống nằm ngoài phạm vi của mã tùy chỉnh đã được triển khai.

Một cách tiếp cận hiệu quả hơn là sử dụng **Extension**. **Extension** đóng vai trò như cầu nối giữa một agent và một API thông qua:

1. Hướng dẫn agent cách sử dụng endpoint của API bằng các ví dụ minh họa.
2. Hướng dẫn agent những tham số hoặc đối số nào cần thiết để thực hiện thành công cuộc gọi đến endpoint của API.



Hình 4: Extensions kết nối Agents với các API bên ngoài

- [1] "Phương thức `get_flights` có thể được sử dụng để lấy thông tin chuyến bay mới nhất..."
- [2] "Khi người dùng muốn tìm kiếm chuyến bay, hãy gọi `get_flights` ..."
- [3] "Các đối số đầu vào cho `get_flights` là `arg1` , `arg2` ,..."

Extensions có thể được tạo độc lập với agent, nhưng cần được cung cấp như một phần của cấu hình agent. Agent sẽ sử dụng mô hình và các ví dụ trong thời gian thực để quyết định sử dụng Extension nào, nếu có, để giải quyết truy vấn của người dùng. Điều này làm nổi bật một điểm mạnh của Extensions, đó là các loại ví

dụ tích hợp sẵn (**built-in example types**) cho phép agent tự động chọn Extension phù hợp nhất cho tác vụ.

Hình 5: Mối quan hệ một-nhiều giữa Agents, Extensions và APIs

Hãy nghĩ về điều này theo cách tương tự như khi một nhà phát triển phần mềm quyết định sử dụng endpoint API nào để giải quyết một vấn đề của người dùng. Nếu người dùng muốn đặt vé máy bay, nhà phát triển có thể sử dụng **Google Flights API**. Nếu người dùng muốn biết vị trí quán cà phê gần nhất so với vị trí hiện tại của họ, nhà phát triển có thể sử dụng **Google Maps API**.

Theo cách tương tự, stack agent hoặc mô hình sẽ sử dụng một tập hợp các **Extensions** đã biết để quyết định cái nào là phù hợp nhất với truy vấn của người dùng. Nếu bạn muốn thấy các **Extensions** hoạt động, bạn có thể thử chúng trên ứng dụng **Gemini** bằng cách vào **Settings > Extensions**, sau đó bật bất kỳ Extension nào mà bạn muốn kiểm tra. Ví dụ: bạn có thể bật Extension **Google Flights**, sau đó yêu cầu Gemini: **"Hiển thị cho tôi các chuyến bay từ Austin đến Zurich khởi hành vào thứ Sáu tới."**

Mô hình mô tả cách một agent có thể sử dụng các công cụ như vòng lặp suy luận (**Reasoning Loop**) và mô hình (**Model**) để chọn lựa các Extensions khác nhau:

- **Flights Extension** kết nối với **Flights API**
- **Maps Extension** kết nối với **Maps API**
- **Weather Extension** kết nối với **Weather API**

Sample Extensions (Các Extension Mẫu)

Để đơn giản hóa việc sử dụng
Extensions, Google cung cấp một số extension có sẵn (**out of the box**

extensions) mà bạn có thể dễ dàng nhập vào dự án của mình và sử dụng với cấu hình tối thiểu. Ví dụ: **Code Interpreter** là một extension trong **Snippet 1** cho phép bạn tạo và chạy mã Python từ mô tả bằng ngôn ngữ tự nhiên.

```
import vertexai
import pprint

PROJECT_ID = "YOUR_PROJECT_ID"
REGION = "us-central1"
vertexai.init(project=PROJECT_ID, location=REGION)

from vertexai.preview.extensions import Extension

extension_code_interpreter = Extension.from_hub("code_interpreter")
CODE_QUERY = """Write a python method to invert a binary tree in O(n) time."""

response = extension_code_interpreter.execute(
operation_id = "generate_and_execute",
operation_params = {"query": CODE_QUERY}
)

print("Generated Code:")
pprint.pprint({response['generated_code']})

# The above snippet will generate the following code.
```

```
Generated Code:
class TreeNode:
def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right
```

```
def invert_binary_tree(root):
    """
```

```
    Đảo ngược một cây nhị phân.
    Args:
        root: Gốc của cây nhị phân.
    Returns:
        Gốc của cây nhị phân đã bị đảo ngược.
    """
    if not root:
        return None

    # Đổi chỗ các con trái và phải đệ quy
    root.left, root.right = invert_binary_tree(root.right), invert_binary_tree(root.left)

    return root

# Ví dụ sử dụng:
# Xây dựng một cây nhị phân mẫu
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

# Đảo ngược cây nhị phân
inverted_root = invert_binary_tree(root)
```

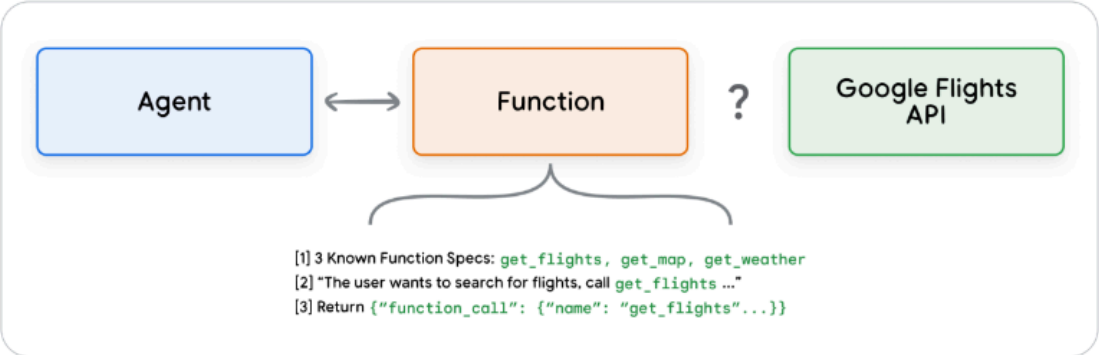
Nói tóm lại, **Extensions** cung cấp một cách để các agent nhận thức, tương tác và ảnh hưởng đến thế giới bên ngoài theo nhiều cách khác nhau. Việc lựa chọn và kích hoạt các Extensions này được hướng dẫn thông qua việc sử dụng các ví dụ (**Examples**), tất cả đều được định nghĩa như một phần của cấu hình Extension.

Functions (Hàm)

Trong lĩnh vực kỹ thuật phần mềm,
hàm (functions) được định nghĩa là các mô-đun mã tự chứa, thực hiện một nhiệm vụ cụ thể và có thể được tái sử dụng khi cần thiết. Khi một nhà phát triển phần mềm viết chương trình, họ thường tạo ra nhiều hàm để thực hiện các tác vụ khác nhau. Họ cũng sẽ xác định logic khi nào cần gọi `function_a` hay `function_b`, cũng như đầu vào và đầu ra mong đợi.

Các hàm hoạt động rất giống trong thế giới của agent, nhưng thay vì nhà phát triển phần mềm, chúng ta có thể thay thế bằng một mô hình AI. Mô hình này có thể nhận một tập hợp các hàm đã biết và quyết định khi nào cần sử dụng từng hàm và những đối số nào mà hàm cần dựa trên mô tả của nó. Các hàm khác với Extensions ở một vài điểm, đáng chú ý là:

1. Mô hình sẽ xuất ra một hàm và các đối số của nó, nhưng không thực hiện cuộc gọi API trực tiếp.
2. Các hàm được thực thi ở phía **client-side**, trong khi Extensions được thực thi ở phía **agent-side**.
   Hình 7: Làm thế nào các hàm (
   Functions) tương tác với các API bên ngoài?



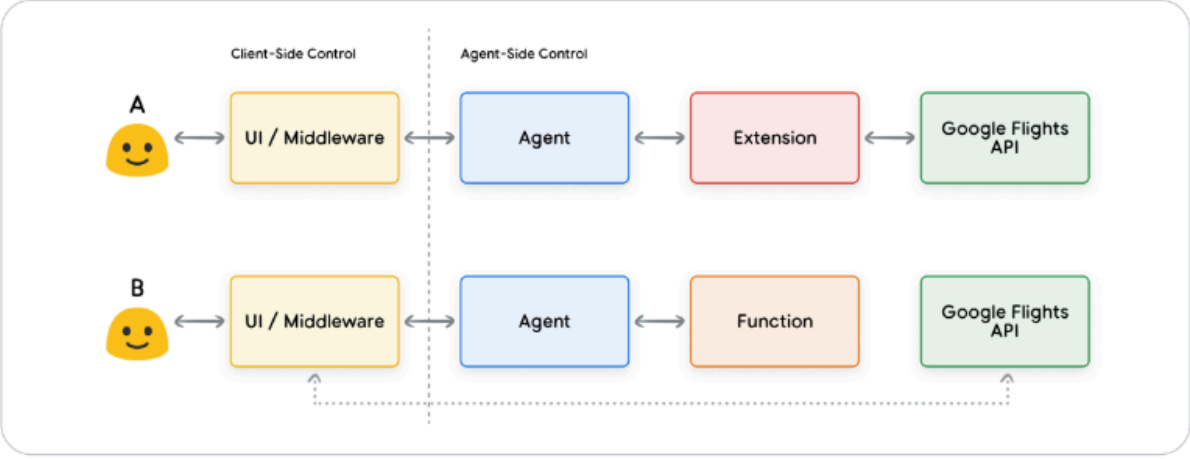
Hình 7: Làm thế nào các hàm (**Functions**) tương tác với các API bên ngoài?

Lưu ý rằng điểm khác biệt chính ở đây là **cả Function và agent đều không tương tác trực tiếp** với Google Flights API. Vậy làm thế nào để cuộc gọi API được thực hiện?

Với các hàm (**Functions**), logic và việc thực hiện cuộc gọi đến endpoint API thực tế được chuyển từ phía agent sang phía ứng dụng client-side, như được mô tả trong Hình 8 và Hình 9 bên dưới. Điều này mang lại cho nhà phát triển khả năng kiểm soát chi tiết hơn đối với luồng dữ liệu trong ứng dụng. Có nhiều lý do mà một nhà phát triển có thể chọn sử dụng các function thay vì sử dụng Extensions, một vài trường hợp phổ biến bao gồm:

- **Cuộc gọi API cần được thực hiện ở một lớp khác của ngăn xếp ứng dụng**, nằm ngoài kiến trúc của agent (ví dụ: hệ thống middleware, framework phía front-end, v.v.).
- **Các hạn chế về bảo mật hoặc xác thực** ngăn agent thực hiện cuộc gọi API trực tiếp (ví dụ: API không được tiếp xúc với Internet hoặc không thể truy cập được từ hạ tầng agent).
- **Các yêu cầu về thời gian hoặc thứ tự thực hiện** khiến agent không thể thực hiện cuộc gọi API theo thời gian thực (ví dụ: xử lý theo lô, đánh giá có sự can thiệp của con người, v.v.).
- **Logic chuyển đổi dữ liệu bổ sung cần được áp dụng cho phản hồi từ API mà agent không thể thực hiện được.** Ví dụ: xem xét một endpoint API không cung cấp cơ chế lọc để giới hạn số lượng kết quả trả về. Việc sử dụng **Functions** ở phía client-side sẽ mang lại cho nhà phát triển nhiều cơ hội hơn để thực hiện các chuyển đổi này.
- **Nhà phát triển muốn cải tiến agent mà không phải triển khai thêm hạ tầng cho các endpoint API** (ví dụ: **Function Calling** có thể hoạt động giống như một phương pháp “stub” cho các API).

Mặc dù sự khác biệt về kiến trúc nội bộ giữa hai cách tiếp cận là tinh tế, như được thấy trong
Hình 8, nhưng khả năng kiểm soát bổ sung và việc giảm thiểu sự phụ thuộc vào hạ tầng bên ngoài làm cho **Function Calling** trở thành một lựa chọn hấp dẫn cho nhà phát triển.



Hình 8: Phân biệt kiểm soát phía client và phía agent cho Extensions và Function Calling

Use cases (Trường hợp sử dụng):

Một mô hình có thể được sử dụng để gọi các hàm (**Functions**) nhằm xử lý các luồng thực thi phức tạp phía client-side cho người dùng cuối, khi mà nhà phát triển agent không muốn mô hình ngôn ngữ kiểm soát việc thực hiện API (như trong trường hợp với Extensions). Hãy xem xét ví dụ sau, khi một agent được huấn luyện như một **trợ lý du lịch**, tương tác với người dùng muốn đặt các chuyến đi nghỉ dưỡng. Mục tiêu là để agent tạo ra một danh sách các thành phố mà chúng ta có thể sử dụng trong ứng dụng middleware của mình để tải về hình ảnh, dữ liệu, v.v., phục vụ cho việc lập kế hoạch chuyến đi của người dùng.

Giả sử người dùng nói như sau:

"Tôi muốn đi nghỉ trượt tuyết với gia đình, nhưng tôi không chắc nên đi đâu."

Với một prompt thông thường đến mô hình, đầu ra có thể như sau:

"Chắc chắn rồi, đây là danh sách các thành phố mà bạn có thể cân nhắc cho chuyến đi trượt tuyết gia đình:"

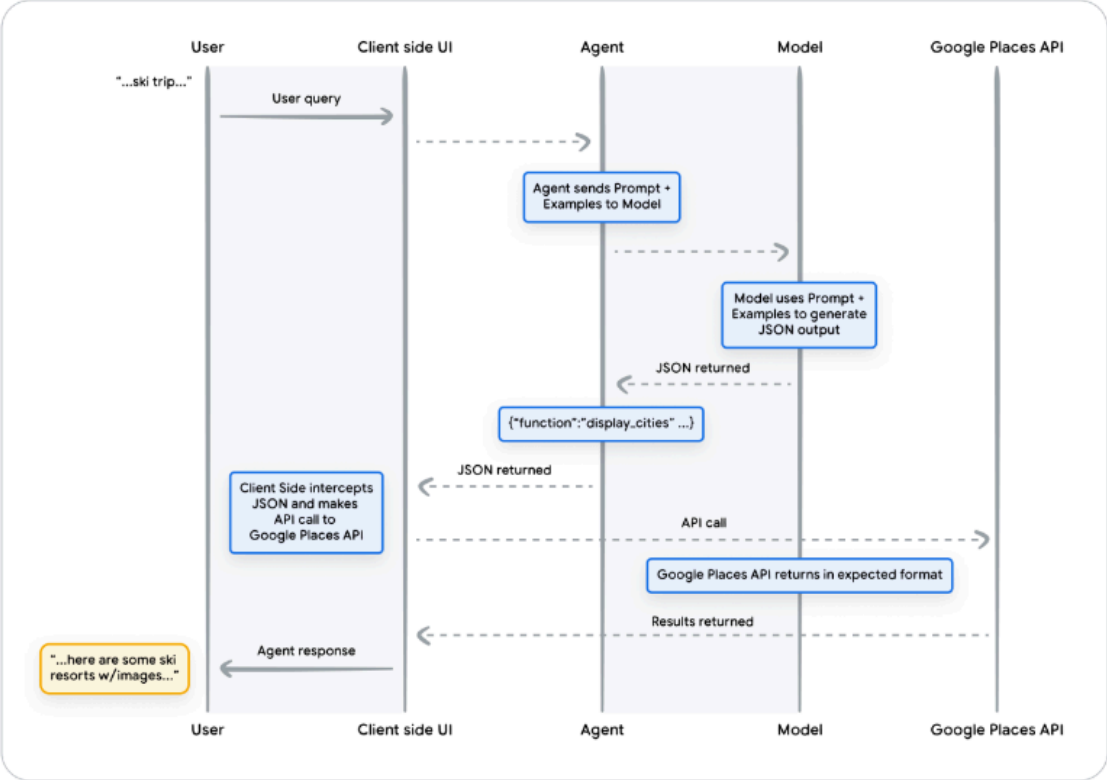
- Crested Butte, Colorado, USA
- Whistler, BC, Canada

- Zermatt, Switzerland

Mặc dù đầu ra trên chứa thông tin mà chúng ta cần (tên các thành phố), nhưng định dạng này không lý tưởng để phân tích cú pháp. Với **Function Calling**, chúng ta có thể dạy cho mô hình định dạng đầu ra theo kiểu cấu trúc (như JSON), giúp hệ thống khác dễ dàng phân tích. Với cùng một prompt đầu vào từ người dùng, ví dụ một đầu ra JSON từ một hàm có thể trông giống như **Snippet 5**.

```
Unset

function_call {
   name: "display_cities",
   args: {
      "cities": ["Crested Butte", "Whistler", "Zermatt"],
      "preferences": "skiing"
   }
}
```

Payload JSON này được tạo ra bởi mô hình và sau đó được gửi đến server phía client để xử lý theo yêu cầu. Trong trường hợp cụ thể này, chúng ta sẽ gọi **Google Places API** để lấy thông tin về các thành phố được cung cấp bởi mô hình, tra cứu hình ảnh, sau đó cung cấp cho người dùng nội dung phong phú đã được định dạng. Hãy xem xét sơ đồ trình tự (sequence diagram) trong **Hình 9**, minh họa chi tiết từng bước của tương tác ở trên:



Hình 9: Sơ đồ trình tự mô tả vòng đời của một Function Call

Kết quả của ví dụ trong Hình 9 là mô hình được sử dụng để "điền vào các chỗ trống" với các tham số cần thiết để
UI phía client có thể thực hiện cuộc gọi đến **Google Places API**. **UI phía client** quản lý cuộc gọi API thực tế bằng cách sử dụng các tham số được cung cấp bởi mô hình trong Function Call đã trả về. Đây chỉ là một trường hợp sử dụng của **Function Calling**, nhưng có nhiều kịch bản khác để cân nhắc, chẳng hạn như:

- Bạn muốn mô hình ngôn ngữ đề xuất một hàm mà bạn có thể sử dụng trong mã, nhưng bạn không muốn bao gồm thông tin xác thực trong mã của mình. Bởi vì **Function Calling** không thực hiện trực tiếp hàm, bạn không cần phải bao gồm thông tin xác thực trong mã cùng với thông tin về hàm.
- **Bạn đang thực hiện các thao tác không đồng bộ (asynchronous)** có thể kéo dài hơn vài giây. Những kịch bản như vậy phù hợp với **Function Calling** vì nó là một thao tác không đồng bộ (asynchronous).

- **Bạn muốn chạy các hàm trên một thiết bị khác** với hệ thống tạo ra các lệnh gọi hàm và các đối số.

Một điểm quan trọng cần nhớ về các hàm (
Functions) là chúng được thiết kế để cung cấp cho nhà phát triển nhiều khả năng kiểm soát hơn, không chỉ đối với việc thực thi các cuộc gọi API mà còn với toàn bộ luồng dữ liệu trong ứng dụng. Trong ví dụ ở **Hình 9**, nhà phát triển đã chọn không trả lại thông tin API cho agent vì nó không liên quan đến các hành động tiếp theo mà agent có thể thực hiện. Tuy nhiên, dựa trên kiến trúc của ứng dụng, việc trả lại dữ liệu từ cuộc gọi API bên ngoài cho agent có thể có ý nghĩa để ảnh hưởng đến suy luận, logic và lựa chọn hành động trong tương lai. Cuối cùng, quyết định là của nhà phát triển ứng dụng để chọn điều gì là phù hợp cho ứng dụng cụ thể.

Function sample code (Mã mẫu cho hàm)

Để đạt được đầu ra như ví dụ trong kịch bản kỳ nghỉ trượt tuyết, chúng ta sẽ xây dựng từng thành phần để làm việc với mô hình **gemini-1.5-flash-001**.

Trước tiên, chúng ta sẽ định nghĩa hàm **display_cities** dưới dạng một phương thức Python đơn giản.

```
from typing import List, Optional

def display_cities(cities: List[str], preferences: Optional[str] = None):
    """
    Provides a list of cities based on the user's search query and preferences.

    Args:
        preferences (str): The user's preferences for the search, like skiing, beach,
        cities (list[str]): The list of cities being recommended to the user.

    Returns:
        list[str]: The list of cities being recommended to the user.
```

```
    """
    return cities
```

```
```

Bước tiếp theo, chúng ta sẽ khởi tạo mô hình, xây dựng công cụ (**Tool**), và truyền truy vấn của người dùng cùng với các công cụ vào mô hình. Việc thực thi mã sẽ tạo ra đầu ra như mô tả ở phần cuối của mã mẫu.

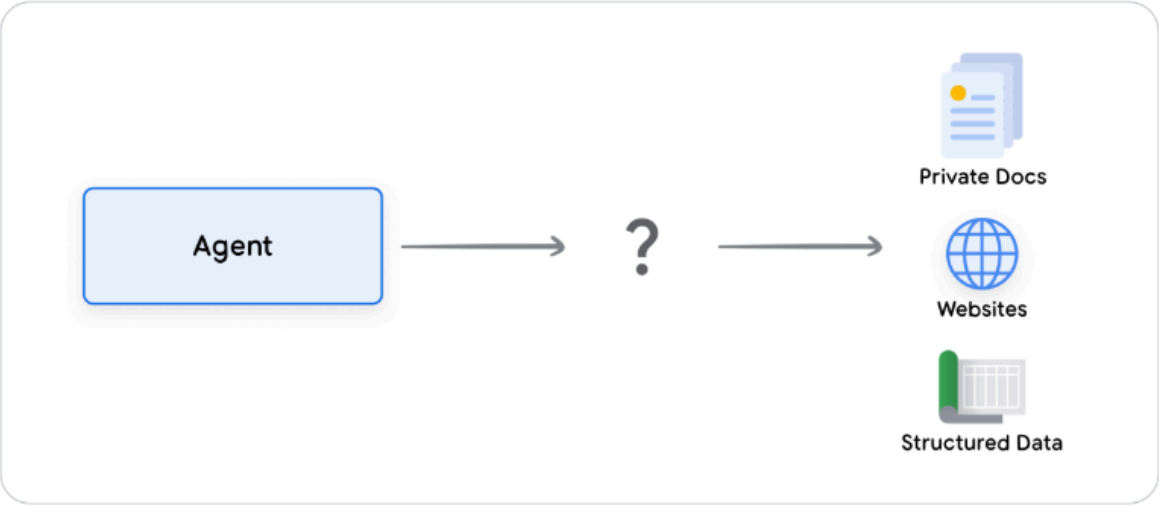
```
from vertexai.generative_models import GenerativeModel, Tool, FunctionDecl
aration

# Khởi tạo mô hình
model = GenerativeModel("gemini-1.5-flash-001")

# Khai báo hàm display_cities
display_cities_function = FunctionDeclaration.from_func(display_cities)
tool = Tool(function_declarations=[display_cities_function])

# Truy vấn của người dùng
message = "Tôi muốn đi nghỉ trượt tuyết với gia đình nhưng tôi không chắc nê
n đi đâu."

# Gửi truy vấn đến mô hình và cho phép lệnh gọi hàm diễn ra
res = model.generate_content(message, tools=[tool])

# In ra tên hàm và các đối số của lệnh gọi hàm
print(f"Function Name: {res.candidates[0].content.parts[0].function_call.nam
e}")
print(f"Function Args: {res.candidates[0].content.parts[0].function_call.arg
s}")

# Kết quả đầu ra:
# > Function Name: display_cities
# > Function Args: {'preferences': 'skiing', 'cities': ['Aspen', 'Vail', 'Park City']}
```

Tóm tắt, f**unctions** cung cấp một framework đơn giản, cho phép nhà phát triển kiểm soát chi tiết luồng dữ liệu và thực thi hệ thống, đồng thời tận dụng mô hình/agent để tạo ra các đầu vào quan trọng. Nhà phát triển có thể chọn trả về dữ liệu từ các cuộc gọi API cho agent hoặc không, tùy thuộc vào yêu cầu kiến trúc của ứng dụng.

Data stores (Kho dữ liệu)

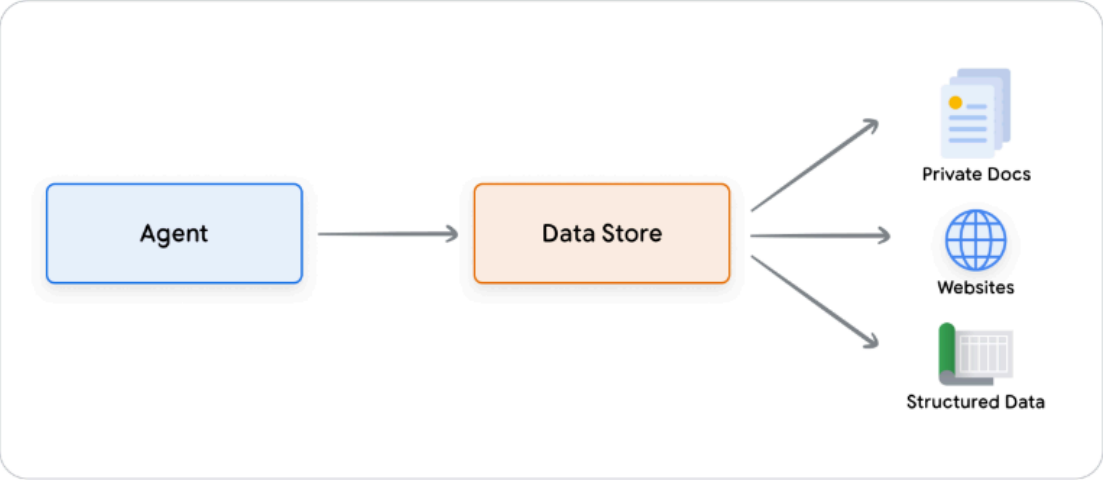
Hãy tưởng tượng một mô hình ngôn ngữ giống như một thư viện khổng lồ chứa dữ liệu huấn luyện của nó, tương tự như một bộ sưu tập sách. Nhưng khác với một thư viện liên tục nhận thêm sách mới, mô hình này không thay đổi, chỉ chứa kiến thức mà nó đã được huấn luyện ban đầu. Điều này tạo ra một thách thức, vì kiến thức thực tế ngoài đời luôn thay đổi. **Data Stores** khắc phục hạn chế này bằng cách cung cấp quyền truy cập đến thông tin linh hoạt và cập nhật hơn, giúp đảm bảo rằng phản hồi của mô hình luôn chính xác và phù hợp với ngữ cảnh hiện tại.

Hãy xem xét một kịch bản phổ biến khi nhà phát triển có thể cần cung cấp một lượng dữ liệu bổ sung nhỏ cho mô hình, có thể dưới dạng bảng tính (**spreadsheets**) hoặc PDF.



Hình 10: Làm thế nào để các agent tương tác với dữ liệu có cấu trúc và không có cấu trúc?

Data Stores cho phép nhà phát triển cung cấp dữ liệu bổ sung ở định dạng gốc cho agent, loại bỏ nhu cầu chuyển đổi dữ liệu phức tạp, huấn luyện lại mô hình hoặc tinh chỉnh mô hình. **Data Store** chuyển đổi tài liệu đến thành một tập hợp các **vector embeddings trong cơ sở dữ liệu** mà agent có thể sử dụng để trích xuất thông tin cần thiết, hỗ trợ cho hành động hoặc phản hồi tiếp theo cho người dùng.



Hình 11: Data Stores kết nối Agents với các nguồn dữ liệu thời gian thực đa dạng

Implementation and application (Triển khai và ứng dụng)

Trong bối cảnh của các agent AI tạo sinh (**Generative AI agents**), **Data Stores** thường được triển khai dưới dạng **cơ sở dữ liệu vector (vector database)** mà nhà phát triển muốn agent truy cập trong thời gian thực. Mặc dù chúng ta sẽ không đi sâu vào cơ sở dữ liệu vector ở đây, điểm chính cần hiểu là chúng lưu trữ dữ liệu dưới dạng **vector embeddings**, một loại vector đa chiều hoặc biểu diễn toán học của dữ liệu đã cung cấp. Một trong những ví dụ phổ biến nhất về việc sử dụng **Data Stores** trong thời gian gần đây là việc triển khai **Retrieval Augmented Generation (RAG)**, một phương pháp tăng cường tìm kiếm thông tin trong mô hình ngôn ngữ. Những ứng dụng này nhằm mở rộng chiều sâu và phạm vi kiến thức của mô hình vượt xa dữ liệu huấn luyện ban đầu bằng cách cho phép mô hình truy cập dữ liệu từ nhiều nguồn khác nhau, bao gồm:

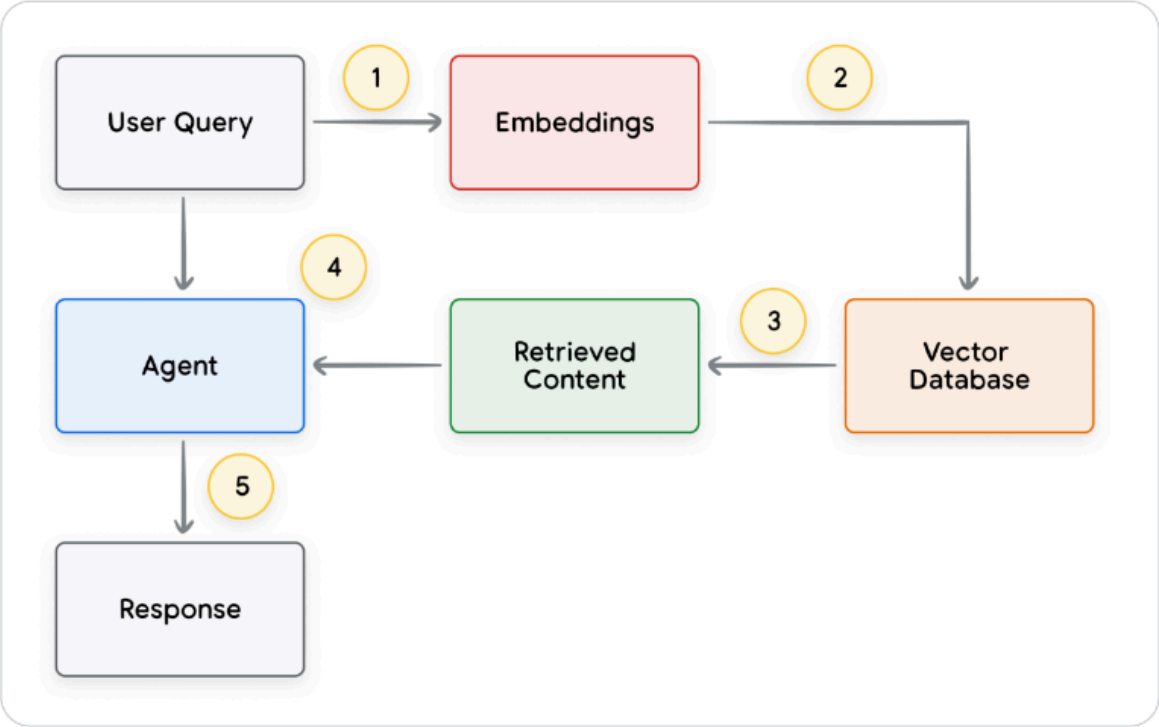
- **Nội dung từ website**
- **Dữ liệu có cấu trúc**, như PDF, tài liệu Word, CSV, bảng tính, v.v.

- **Dữ liệu không có cấu trúc**, như HTML, PDF, TXT, v.v.

Hình 12: Mối quan hệ một-nhiều giữa agent và các Data Store, có thể đại diện cho các loại dữ liệu đã được lập chỉ mục trước

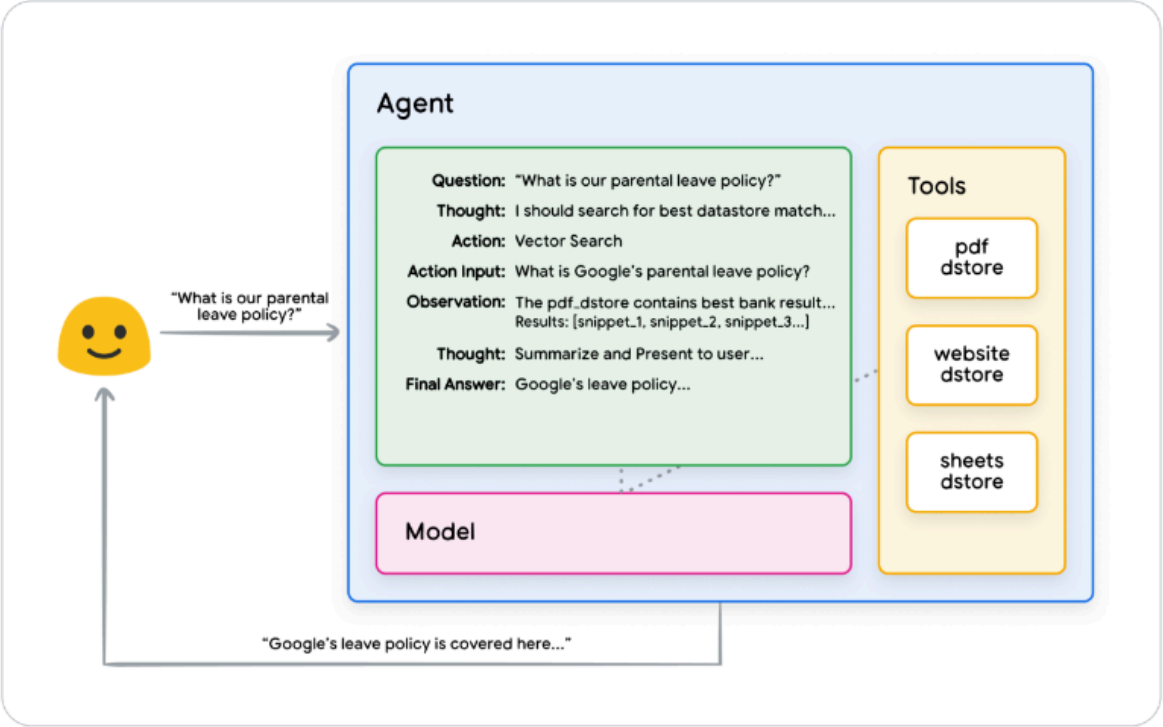
Quy trình cơ bản cho mỗi truy vấn của người dùng và vòng lặp phản hồi của agent thường được mô hình hóa như sau (xem Hình 13):

1. **Truy vấn của người dùng** được gửi đến một mô hình embedding để tạo các embeddings cho truy vấn.
2. **Các embeddings của truy vấn** sau đó được so khớp với nội dung của cơ sở dữ liệu vector bằng cách sử dụng một thuật toán so khớp, ví dụ như **SCaNN**.
3. **Nội dung phù hợp** được truy xuất từ cơ sở dữ liệu vector dưới dạng văn bản và được gửi lại cho agent.
4. **Agent nhận được** cả truy vấn của người dùng và nội dung đã truy xuất, sau đó tạo ra phản hồi hoặc hành động.
5. Agent nhận nội dung đã truy xuất, tạo ra phản hồi cuối cùng và gửi cho người dùng.



Hình 13: Vòng đời của truy vấn người dùng và phản hồi của agent trong ứng dụng dựa trên RAG

Kết quả cuối cùng là một ứng dụng cho phép agent so khớp truy vấn của người dùng với kho dữ liệu đã biết thông qua tìm kiếm vector, truy xuất nội dung gốc, và cung cấp nội dung này cho lớp điều phối (**orchestration layer**) và mô hình để xử lý tiếp theo. Hành động tiếp theo có thể là đưa ra câu trả lời cuối cùng cho người dùng hoặc thực hiện một lần tìm kiếm vector bổ sung để tinh chỉnh kết quả hơn nữa. Một ví dụ tương tác với agent sử dụng **RAG với ReAct reasoning/planning** sẽ được minh họa trong **Hình 14**.



Hình 14: Ví dụ về ứng dụng dựa trên RAG với lập luận và lập kế hoạch ReAct

Tools Recap (Tóm tắt về công cụ)

Bảng dưới đây tóm tắt các loại công cụ bao gồm **Extensions**, **Function Calling**, và **Data Stores**. Mỗi loại công cụ có mục đích riêng và có thể được sử dụng riêng lẻ hoặc kết hợp tùy thuộc vào quyết định của nhà phát triển agent.

Công cụ	Extensions	Function Calling	Data Stores
Thực thi	Agent-Side Execution	Client-Side Execution	Agent-Side Execution
Trường hợp sử dụng	- Nhà phát triển muốn agent kiểm soát tương tác với các endpoint API. - Hữu ích khi tận dụng các Extension có sẵn	- Các hạn chế về bảo mật hoặc xác thực ngăn agent thực hiện cuộc gọi API trực tiếp. - Các ràng buộc về thời gian hoặc thứ tự thực	Nhà phát triển muốn triển khai **Retrieval Augmented Generation (RAG)** với các loại dữ liệu sau:

	(như Vertex Search, Code Interpreter, v.v.). - Lập kế hoạch đa bước (multi-hop) và thực hiện cuộc gọi API (hành động tiếp theo của agent phụ thuộc vào kết quả của hành động trước đó/API call).	hiện ngăn agent thực hiện cuộc gọi API theo thời gian thực (ví dụ: xử lý theo lô, đánh giá có sự can thiệp của con người, v.v.). - API không được kết nối với Internet hoặc không thể truy cập được bởi hệ thống của Google.	- Nội dung từ trang web hoặc URL đã được lập chỉ mục trước. - Dữ liệu có cấu trúc như PDF, tài liệu Word, CSV, bảng tính, v.v. - Cơ sở dữ liệu quan hệ/phi quan hệ. - Dữ liệu không có cấu trúc như HTML, PDF, TXT, v.v.

Enhancing model performance with targeted learning (Nâng cao hiệu suất của mô hình với học tập có mục tiêu)

Một khía cạnh quan trọng của việc sử dụng mô hình hiệu quả là khả năng chọn công cụ phù hợp khi tạo ra kết quả, đặc biệt khi sử dụng các công cụ ở quy mô lớn trong môi trường sản xuất. Trong khi việc huấn luyện tổng quát giúp mô hình phát triển kỹ năng này, các tình huống thực tế thường yêu cầu kiến thức vượt ra ngoài dữ liệu huấn luyện ban đầu. Hãy tưởng tượng đây là sự khác biệt giữa kỹ năng nấu ăn cơ bản và việc thành thạo một món ăn đặc trưng. Cả hai đều cần kiến thức nền tảng về nấu ăn, nhưng việc thành thạo yêu cầu học tập có mục tiêu để đạt được kết quả tinh tế hơn. Để giúp mô hình truy cập loại kiến thức đặc thù này, có một số phương pháp tồn tại:

- **In-context learning (Học trong ngữ cảnh)**: Phương pháp này cung cấp cho mô hình tổng quát một prompt, công cụ, và các ví dụ "few-shot" tại thời điểm suy luận, cho phép mô hình học cách sử dụng các công cụ cho một nhiệm vụ

cụ thể. **ReAct framework** là một ví dụ về phương pháp này trong ngôn ngữ tự nhiên.

- **Retrieval-based in-context learning (Học trong ngữ cảnh dựa trên truy xuất thông tin)**: Kỹ thuật này sẽ tự động điền vào prompt của mô hình với thông tin, công cụ và các ví dụ liên quan nhất bằng cách truy xuất từ bộ nhớ ngoài. Một ví dụ về điều này là **Example Store** trong các tiện ích mở rộng của **Vertex AI** hoặc kiến trúc **Data Store** dựa trên RAG đã được đề cập trước đó.
- **Fine-tuning based learning (Học dựa trên tinh chỉnh)**: Phương pháp này liên quan đến việc huấn luyện mô hình bằng một bộ dữ liệu lớn hơn với các ví dụ cụ thể trước khi suy luận. Điều này giúp mô hình hiểu cách thức và thời điểm áp dụng các công cụ nhất định trước khi nhận được truy vấn từ người dùng.

Để cung cấp thêm cái nhìn sâu sắc về từng phương pháp học tập có mục tiêu, hãy quay lại ví dụ minh họa về nấu ăn.

- Hãy tưởng tượng một đầu bếp nhận được một công thức cụ thể (chính là prompt), một vài nguyên liệu chính (các công cụ liên quan), và một số món ăn mẫu (ví dụ few-shot) từ khách hàng. Dựa trên thông tin hạn chế này và kiến thức chung của đầu bếp về nấu ăn, họ sẽ cần phải tìm ra cách chế biến món ăn "ngay lập tức" sao cho phù hợp nhất với công thức và sở thích của khách hàng. Đây chính là **học trong ngữ cảnh (in-context learning)**.
- Bây giờ, hãy tưởng tượng đầu bếp đang ở trong một nhà bếp có đầy đủ nguyên liệu (external data stores) chứa nhiều loại nguyên liệu và sách nấu ăn (các ví dụ và công cụ). Lúc này, đầu bếp có thể chọn các nguyên liệu và công thức nấu ăn một cách linh hoạt từ kho dữ liệu và điều chỉnh tốt hơn theo công thức và sở thích của khách hàng. Điều này cho phép đầu bếp tạo ra món ăn có thông tin phong phú hơn, tận dụng cả kiến thức hiện có và kiến thức mới. Đây là **học trong ngữ cảnh dựa trên truy xuất thông tin (retrieval-based in-context learning)**.
- Cuối cùng, hãy tưởng tượng rằng chúng ta đưa đầu bếp trở lại trường học để học một nền ẩm thực mới hoặc một loạt món ăn mới (tiền huấn luyện trên một bộ dữ liệu lớn với các ví dụ cụ thể). Điều này cho phép đầu bếp hiểu sâu hơn khi phải tiếp cận các công thức mới chưa từng thấy từ khách hàng. Cách tiếp cận này hoàn hảo nếu chúng ta muốn đầu bếp xuất sắc trong các nền ẩm thực

cụ thể (các lĩnh vực kiến thức). Đây chính là **học dựa trên tinh chỉnh (fine-tuning based learning)**.

Mỗi phương pháp trên đều mang lại những ưu điểm và nhược điểm riêng về tốc độ, chi phí, và độ trễ. Tuy nhiên, bằng cách kết hợp những kỹ thuật này trong một framework agent, chúng ta có thể tận dụng các điểm mạnh và giảm thiểu các điểm yếu, tạo ra một giải pháp linh hoạt và bền vững hơn.

Agent quick start with LangChain (Bắt đầu nhanh với agent sử dụng LangChain)

Để cung cấp một ví dụ thực tế về cách một agent có thể hoạt động, chúng ta sẽ xây dựng một nguyên mẫu nhanh bằng cách sử dụng các thư viện **LangChain** và **LangGraph**. Đây là những thư viện mã nguồn mở phổ biến, cho phép người dùng xây dựng các agent tùy chỉnh bằng cách **"kết chuỗi"** các logic, lập luận, và các lệnh gọi công cụ để trả lời truy vấn của người dùng. Chúng ta sẽ sử dụng mô hình **gemini-1.5-flash-001** và một số công cụ đơn giản để trả lời một truy vấn nhiều bước từ người dùng, như được mô tả trong **Snippet 8**

Các công cụ mà chúng ta sẽ sử dụng bao gồm:

- **SerpAPI** (dùng cho tìm kiếm Google)
- **Google Places API**

Sau khi chạy chương trình trong **Snippet 8**, bạn có thể xem kết quả đầu ra mẫu trong **Snippet 9**.

```
from langgraph.prebuilt import create_react_agent
from langchain.core.tools import tool
from langchain_community.utilities import SerpAPIWrapper
from langchain_community.tools import GooglePlacesTool

# Đặt API key cho SerpAPI và Google Places API
os.environ["SERPAPI_API_KEY"] = "XXXXX"
os.environ["GPLACES_API_KEY"] = "XXXXX"
```

```
# Định nghĩa công cụ tìm kiếm với SerpAPI
@tool
def search(query: str):
    """Sử dụng SerpAPI để thực hiện tìm kiếm Google."""
    search = SerpAPIWrapper()
    return search.run(query)

# Định nghĩa công cụ tra cứu địa điểm với Google Places API
@tool
def places(query: str):
    """Sử dụng Google Places API để thực hiện truy vấn địa điểm."""
    places = GooglePlacesTool()
    return places.run(query)

# Khởi tạo mô hình ChatVertexAI
model = ChatVertexAI(model="gemini-1.5-flash-001")
tools = [search, places]

# Truy vấn của người dùng
query = "Who did the Texas Longhorns play in football last week? What is the ad
of the other team's stadium?"

# Tạo agent với mô hình và các công cụ
agent = create_react_agent(model, tools)
input = {"messages": [("human", query)]}

for s in agent.stream(input, stream_mode="values"):
    message = s["messages"][-1]
    if isinstance(message, tuple):
        print(message)
    else:
        message.pretty_print()
```

```
============================ Human Message ==================
Who did the Texas Longhorns play in football last week? What is the address of t
============================ Ai Message ====================
Tool Calls: search
Args:
   query: Texas Longhorns football schedule
============================ Tool Message ===================
Name: search
{...Results: "NCAA Division I Football, Georgia, Date..."}
============================ Ai Message ====================
The Texas Longhorns played the Georgia Bulldogs last week.
Tool Calls: places
Args:
   query: Georgia Bulldogs stadium
============================ Tool Message ===================
Name: places
{...Sanford Stadium Address: 100 Sanford...}
============================ Ai Message ====================
The address of the Georgia Bulldogs stadium is 100 Sanford Dr, Athens, GA 3060
```

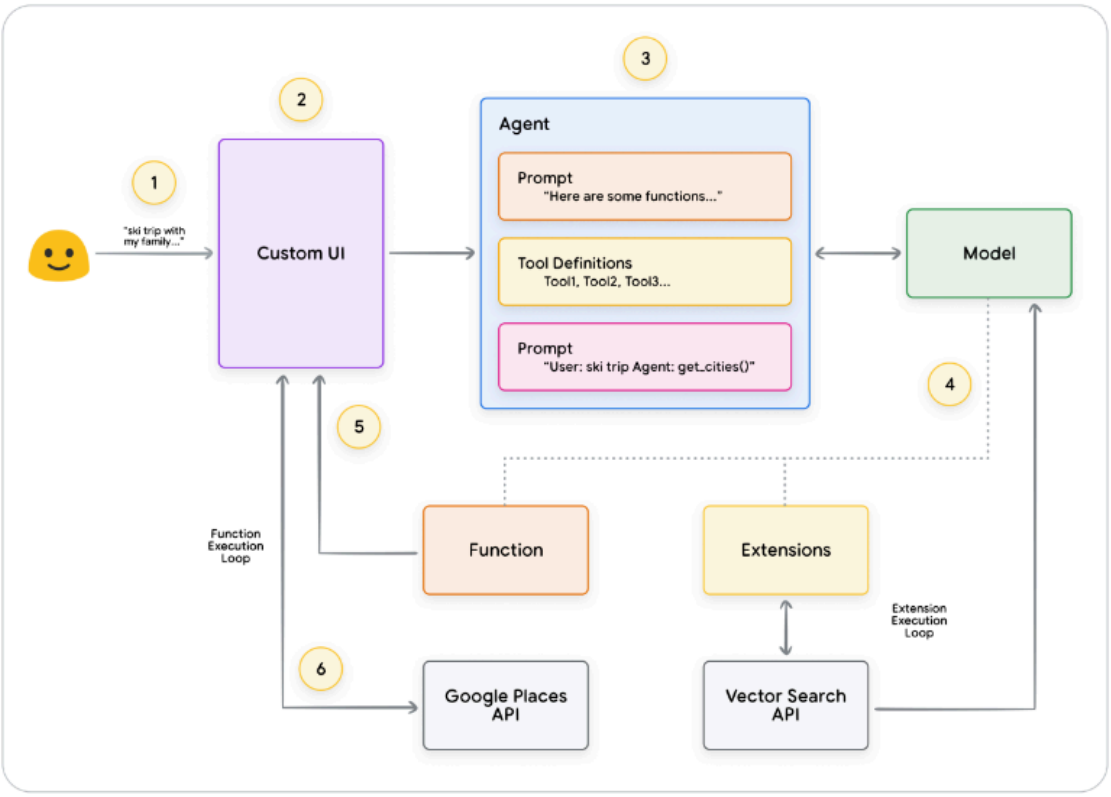
Mặc dù đây chỉ là một ví dụ đơn giản về agent, nhưng nó minh họa các thành phần cơ bản của
Mô hình, **Điều phối (Orchestration)** và **các công cụ (Tools)** cùng làm việc để đạt được một mục tiêu cụ thể. Trong phần cuối, chúng ta sẽ khám phá cách những thành phần này kết hợp với nhau trong các sản phẩm quản lý quy mô lớn của Google như **Vertex AI agents** và **Generative Playbooks**.

Production applications with Vertex AI agents (Ứng dụng sản xuất với các agent của Vertex AI)

Mặc dù tài liệu này đã khám phá các thành phần cốt lõi của agent, việc xây dựng các ứng dụng cấp sản xuất yêu cầu tích hợp chúng với các công cụ bổ sung như

giao diện người dùng, framework đánh giá, và cơ chế cải tiến liên tục. Nền tảng **Vertex AI** của Google giúp đơn giản hóa quy trình này bằng cách cung cấp một môi trường quản lý hoàn toàn với tất cả các thành phần cơ bản đã được đề cập trước đó. Sử dụng **giao diện ngôn ngữ tự nhiên (natural language interface)**, các nhà phát triển có thể nhanh chóng xác định các yếu tố quan trọng của agent - mục tiêu, hướng dẫn nhiệm vụ, công cụ, các sub-agent cho việc phân công nhiệm vụ, và các ví dụ - để dễ dàng xây dựng hành vi hệ thống mong muốn. Ngoài ra, nền tảng này đi kèm với một bộ công cụ phát triển cho phép thử nghiệm, đánh giá, đo lường hiệu suất agent, gỡ lỗi và cải thiện chất lượng tổng thể của các agent được phát triển. Điều này cho phép nhà phát triển tập trung vào việc xây dựng và tinh chỉnh các agent của họ, trong khi các phức tạp về hạ tầng, triển khai, và bảo trì được nền tảng tự quản lý.

Trong **Hình 15**, chúng tôi đã cung cấp một kiến trúc mẫu của một agent được xây dựng trên nền tảng **Vertex AI**, sử dụng các tính năng khác nhau như **Vertex Agent Builder**, **Vertex Extensions**, **Vertex Function Calling**, và **Vertex Example Store**, chỉ để kể ra một số ví dụ. Kiến trúc này bao gồm nhiều thành phần cần thiết cho một ứng dụng sẵn sàng cho sản xuất.



Hình 15: Kiến trúc mẫu end-to-end của agent được xây dựng trên nền tảng Vertex AI

Bạn có thể thử một mẫu kiến trúc tác nhân đã được xây dựng sẵn từ tài liệu chính thức của chúng tôi.

Summary (Tóm tắt)

Trong tài liệu này, chúng tôi đã thảo luận về các khối xây dựng nền tảng của các agent AI tạo sinh, cấu trúc của chúng và các cách hiệu quả để triển khai chúng dưới dạng kiến trúc nhận thức (cognitive). Một số điểm chính từ tài liệu này bao gồm:

1. **Agent mở rộng khả năng của các mô hình ngôn ngữ**:
    - Agent tận dụng các công cụ để truy cập thông tin thời gian thực, đề xuất các hành động trong thế giới thực, lập kế hoạch và thực hiện các nhiệm vụ

phức tạp một cách tự động. Các agent có thể sử dụng một hoặc nhiều mô hình ngôn ngữ để quyết định khi nào và làm thế nào để chuyển đổi trạng thái và sử dụng các công cụ bên ngoài để hoàn thành các nhiệm vụ phức tạp mà mô hình không thể tự hoàn thành.

2. **Lớp điều phối (Orchestration layer) là trung tâm hoạt động của agent**:
    - Tại cốt lõi của hoạt động của agent là lớp điều phối, một kiến trúc nhận thức định hình suy luận, lập kế hoạch, ra quyết định và hướng dẫn các hành động. Các kỹ thuật suy luận như **ReAct**, **Chain-of-Thought**, và **Tree-of-Thoughts** cung cấp một khung làm việc cho lớp điều phối để tiếp nhận thông tin, thực hiện suy luận nội bộ và tạo ra các quyết định hoặc phản hồi có cơ sở.
3. **Công cụ như Extensions, Functions, và Data Stores**:
    - Các công cụ này là chìa khóa kết nối agent với thế giới bên ngoài, cho phép agent tương tác với các hệ thống ngoại vi và truy cập kiến thức vượt ra ngoài dữ liệu huấn luyện ban đầu. **Extensions** cung cấp cầu nối giữa agent và các API bên ngoài, cho phép thực hiện các cuộc gọi API và truy xuất thông tin thời gian thực. **Functions** cung cấp sự kiểm soát tinh tế hơn cho nhà phát triển, cho phép tạo ra các tham số Function có thể được thực thi phía client-side. **Data Stores** cung cấp quyền truy cập vào dữ liệu có cấu trúc và không có cấu trúc, hỗ trợ các ứng dụng dựa trên dữ liệu.

Tương lai của các agent hứa hẹn nhiều tiến bộ thú vị, và chúng ta chỉ mới bắt đầu khám phá bề mặt của những gì có thể thực hiện được. Khi các công cụ trở nên tinh vi hơn và khả năng suy luận được cải thiện, các agent sẽ có khả năng giải quyết các vấn đề ngày càng phức tạp hơn. Hơn nữa, cách tiếp cận chiến lược của **'agent chaining'** (kết chuỗi agent) sẽ tiếp tục phát triển mạnh mẽ. Bằng cách kết hợp các agent chuyên biệt - mỗi agent xuất sắc trong một lĩnh vực hoặc nhiệm vụ cụ thể - chúng ta có thể tạo ra một phương pháp **'mixture of agent experts'** (hỗn hợp các chuyên gia agent), có khả năng mang lại kết quả vượt trội trên nhiều ngành công nghiệp và các lĩnh vực khác nhau.

Điều quan trọng cần nhớ là việc xây dựng các kiến trúc agent phức tạp đòi hỏi một cách tiếp cận lặp đi lặp lại. Thử nghiệm và tinh chỉnh là yếu tố then chốt để tìm ra

các giải pháp cho các trường hợp kinh doanh cụ thể và nhu cầu của tổ chức. Không có hai agent nào giống nhau do bản chất tạo sinh của các mô hình nền tảng làm cơ sở cho kiến trúc của chúng. Tuy nhiên, bằng cách khai thác sức mạnh của từng thành phần nền tảng này, chúng ta có thể tạo ra các ứng dụng có ảnh hưởng, mở rộng khả năng của các mô hình ngôn ngữ và mang lại giá trị thực tế.