**Chapter 2\_ Routing**

Chapter 2: Routing

**Routing Pattern Overview**

While sequential processing via prompt chaining is a foundational technique for executing deterministic, linear workflows with language models, its applicability is limited in scenarios requiring adaptive responses. Real-world agentic systems must often arbitrate between multiple potential actions based on contingent factors, such as the state of the environment, user input, or the outcome of a preceding operation. This capacity for dynamic decision-making, which governs the flow of control to different specialized functions, tools, or sub-processes, is achieved through a mechanism known as routing.

Routing introduces conditional logic into an agent's operational framework, enabling a shift from a fixed execution path to a model where the agent dynamically evaluates specific criteria to select from a set of possible subsequent actions. This allows for more flexible and context-aware system behavior.

For instance, an agent designed for customer inquiries, when equipped with a routing function, can first classify an incoming query to determine the user's intent. Based on this classification, it can then direct the query to a specialized agent for direct question-answering, a database retrieval tool for account information, or an escalation procedure for complex issues, rather than defaulting to a single, predetermined response pathway. Therefore, a more sophisticated agent using routing could:

1. Analyze the user's query.
2. **Route** the query based on its *intent*:

* If the intent is "check order status", route to a sub-agent or tool chain that interacts with the order database.
* If the intent is "product information", route to a sub-agent or chain that searches the product catalog.
* If the intent is "technical support", route to a different chain that accesses troubleshooting guides or escalates to a human.
* If the intent is unclear, route to a clarification sub-agent or prompt chain.

The core component of the Routing pattern is a mechanism that performs the evaluation and directs the flow. This mechanism can be implemented in several ways:

* **LLM-based Routing:** The language model itself can be prompted to analyze the input and output a specific identifier or instruction that indicates the next step or destination. For example, a prompt might ask the LLM to "Analyze the following user query and output only the category: 'Order Status', 'Product Info', 'Technical Support', or 'Other'." The agentic system then reads this output and directs the workflow accordingly.
* **Embedding-based Routing:** The input query can be converted into a vector embedding (see RAG, Chapter 14). This embedding is then compared to embeddings representing different routes or capabilities. The query is routed to the route whose embedding is most similar. This is useful for semantic routing, where the decision is based on the meaning of the input rather than just keywords.
* **Rule-based Routing:** This involves using predefined rules or logic (e.g., if-else statements, switch cases) based on keywords, patterns, or structured data extracted from the input. This can be faster and more deterministic than LLM-based routing, but is less flexible for handling nuanced or novel inputs.
* **Machine Learning Model-Based Routing**: it employs a discriminative model, such as a classifier, that has been specifically trained on a small corpus of labeled data to perform a routing task. While it shares conceptual similarities with embedding-based methods, its key characteristic is the supervised fine-tuning process, which adjusts the model's parameters to create a specialized routing function. This technique is distinct from LLM-based routing because the decision-making component is not a generative model executing a prompt at inference time. Instead, the routing logic is encoded within the fine-tuned model's learned weights. While LLMs may be used in a pre-processing step to generate synthetic data for augmenting the training set, they are not involved in the real-time routing decision itself.

Routing mechanisms can be implemented at multiple junctures within an agent's operational cycle. They can be applied at the outset to classify a primary task, at intermediate points within a processing chain to determine a subsequent action, or during a subroutine to select the most appropriate tool from a given set.

Computational frameworks such as LangChain, LangGraph, and Google's Agent Developer Kit (ADK) provide explicit constructs for defining and managing such conditional logic. With its state-based graph architecture, LangGraph is particularly well-suited for complex routing scenarios where decisions are contingent upon the accumulated state of the entire system. Similarly, Google's ADK provides foundational components for structuring an agent's capabilities and interaction models, which serve as the basis for implementing routing logic. Within the execution environments provided by these frameworks, developers define the possible operational paths and the functions or model-based evaluations that dictate the transitions between nodes in the computational graph.

The implementation of routing enables a system to move beyond deterministic sequential processing. It facilitates the development of more adaptive execution flows that can respond dynamically and appropriately to a wider range of inputs and state changes.

**Practical Applications & Use Cases**

The routing pattern is a critical control mechanism in the design of adaptive agentic systems, enabling them to dynamically alter their execution path in response to variable inputs and internal states. Its utility spans multiple domains by providing a necessary layer of conditional logic.

In human-computer interaction, such as with virtual assistants or AI-driven tutors, routing is employed to interpret user intent. An initial analysis of a natural language query determines the most appropriate subsequent action, whether it is invoking a specific information retrieval tool, escalating to a human operator, or selecting the next module in a curriculum based on user performance. This allows the system to move beyond linear dialogue flows and respond contextually.

Within automated data and document processing pipelines, routing serves as a classification and distribution function. Incoming data, such as emails, support tickets, or API payloads, is analyzed based on content, metadata, or format. The system then directs each item to a corresponding workflow, such as a sales lead ingestion process, a specific data transformation function for JSON or CSV formats, or an urgent issue escalation path.

In complex systems involving multiple specialized tools or agents, routing acts as a high-level dispatcher. A research system composed of distinct agents for searching, summarizing, and analyzing information would use a router to assign tasks to the most suitable agent based on the current objective. Similarly, an AI coding assistant uses routing to identify the programming language and user's intent—to debug, explain, or translate—before passing a code snippet to the correct specialized tool.

Ultimately, routing provides the capacity for logical arbitration that is essential for creating functionally diverse and context-aware systems. It transforms an agent from a static executor of pre-defined sequences into a dynamic system that can make decisions about the most effective method for accomplishing a task under changing conditions.

**Hands-On Code Example (LangChain)**

Implementing routing in code involves defining the possible paths and the logic that decides which path to take. Frameworks like LangChain and LangGraph provide specific components and structures for this. LangGraph's state-based graph structure is particularly intuitive for visualizing and implementing routing logic.

This code demonstrates a simple agent-like system using LangChain and Google's Generative AI. It sets up a "coordinator" that routes user requests to different simulated "sub-agent" handlers based on the request's intent (booking, information, or unclear). The system uses a language model to classify the request and then delegates it to the appropriate handler function, simulating a basic delegation pattern often seen in multi-agent architectures.

First, ensure you have the necessary libraries installed:

|  |
| --- |
| pip install langchain langgraph google-cloud-aiplatform langchain-google-genai google-adk deprecated pydantic |

You will also need to set up your environment with your API key for the language model you choose (e.g., OpenAI, Google Gemini, Anthropic).

|  |
| --- |
| # Copyright (c) 2025 Marco Fago  # https://www.linkedin.com/in/marco-fago/  #  # This code is licensed under the MIT License.  # See the LICENSE file in the repository for the full license text.  from langchain\_google\_genai import ChatGoogleGenerativeAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.output\_parsers import StrOutputParser  from langchain\_core.runnables import RunnablePassthrough, RunnableBranch  # --- Configuration ---  # Ensure your API key environment variable is set (e.g., GOOGLE\_API\_KEY)  try:  llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)  print(f"Language model initialized: {llm.model}")  except Exception as e:  print(f"Error initializing language model: {e}")  llm = None  # --- Define Simulated Sub-Agent Handlers (equivalent to ADK sub\_agents) ---  def booking\_handler(request: str) -> str:  """Simulates the Booking Agent handling a request."""  print("\n--- DELEGATING TO BOOKING HANDLER ---")  return f"Booking Handler processed request: '{request}'. Result: Simulated booking action."  def info\_handler(request: str) -> str:  """Simulates the Info Agent handling a request."""  print("\n--- DELEGATING TO INFO HANDLER ---")  return f"Info Handler processed request: '{request}'. Result: Simulated information retrieval."  def unclear\_handler(request: str) -> str:  """Handles requests that couldn't be delegated."""  print("\n--- HANDLING UNCLEAR REQUEST ---")  return f"Coordinator could not delegate request: '{request}'. Please clarify."  # --- Define Coordinator Router Chain (equivalent to ADK coordinator's instruction) ---  # This chain decides which handler to delegate to.  coordinator\_router\_prompt = ChatPromptTemplate.from\_messages([  ("system", """Analyze the user's request and determine which specialist handler should process it.  - If the request is related to booking flights or hotels,  output 'booker'.  - For all other general information questions, output 'info'.  - If the request is unclear or doesn't fit either category,  output 'unclear'.  ONLY output one word: 'booker', 'info', or 'unclear'."""),  ("user", "{request}")  ])  if llm:  coordinator\_router\_chain = coordinator\_router\_prompt | llm | StrOutputParser()  # --- Define the Delegation Logic (equivalent to ADK's Auto-Flow based on sub\_agents) ---  # Use RunnableBranch to route based on the router chain's output.  # Define the branches for the RunnableBranch  branches = {  "booker": RunnablePassthrough.assign(output=lambda x: booking\_handler(x['request']['request'])),  "info": RunnablePassthrough.assign(output=lambda x: info\_handler(x['request']['request'])),  "unclear": RunnablePassthrough.assign(output=lambda x: unclear\_handler(x['request']['request'])),  }  # Create the RunnableBranch. It takes the output of the router chain  # and routes the original input ('request') to the corresponding handler.  delegation\_branch = RunnableBranch(  (lambda x: x['decision'].strip() == 'booker', branches["booker"]), # Added .strip()  (lambda x: x['decision'].strip() == 'info', branches["info"]), # Added .strip()  branches["unclear"] # Default branch for 'unclear' or any other output  )  # Combine the router chain and the delegation branch into a single runnable  # The router chain's output ('decision') is passed along with the original input ('request')  # to the delegation\_branch.  coordinator\_agent = {  "decision": coordinator\_router\_chain,  "request": RunnablePassthrough()  } | delegation\_branch | (lambda x: x['output']) # Extract the final output  # --- Example Usage ---  def main():  if not llm:  print("\nSkipping execution due to LLM initialization failure.")  return  print("--- Running with a booking request ---")  request\_a = "Book me a flight to London."  result\_a = coordinator\_agent.invoke({"request": request\_a})  print(f"Final Result A: {result\_a}")  print("\n--- Running with an info request ---")  request\_b = "What is the capital of Italy?"  result\_b = coordinator\_agent.invoke({"request": request\_b})  print(f"Final Result B: {result\_b}")  print("\n--- Running with an unclear request ---")  request\_c = "Tell me about quantum physics."  result\_c = coordinator\_agent.invoke({"request": request\_c})  print(f"Final Result C: {result\_c}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

As mentioned, this Python code constructs a simple agent-like system using the LangChain library and Google's Generative AI model, specifically gemini-2.5-flash. In detail, It defines three simulated sub-agent handlers: booking\_handler, info\_handler, and unclear\_handler, each designed to process specific types of requests.

A core component is the coordinator\_router\_chain, which utilizes a ChatPromptTemplate to instruct the language model to categorize incoming user requests into one of three categories: 'booker', 'info', or 'unclear'. The output of this router chain is then used by a RunnableBranch to delegate the original request to the corresponding handler function. The RunnableBranch checks the decision from the language model and directs the request data to either the booking\_handler, info\_handler, or unclear\_handler. The coordinator\_agent combines these components, first routing the request for a decision and then passing the request to the chosen handler. The final output is extracted from the handler's response.

The main function demonstrates the system's usage with three example requests, showcasing how different inputs are routed and processed by the simulated agents. Error handling for language model initialization is included to ensure robustness. The code structure mimics a basic multi-agent framework where a central coordinator delegates tasks to specialized agents based on intent.

**Hands-On Code Example (Google ADK)**

The Agent Development Kit (ADK) is a framework for engineering agentic systems, providing a structured environment for defining an agent's capabilities and behaviours. In contrast to architectures based on explicit computational graphs, routing within the ADK paradigm is typically implemented by defining a discrete set of "tools" that represent the agent's functions. The selection of the appropriate tool in response to a user query is managed by the framework's internal logic, which leverages an underlying model to match user intent to the correct functional handler.

This Python code demonstrates an example of an Agent Development Kit (ADK) application using Google's ADK library. It sets up a "Coordinator" agent that routes user requests to specialized sub-agents ("Booker" for bookings and "Info" for general information) based on defined instructions. The sub-agents then use specific tools to simulate handling the requests, showcasing a basic delegation pattern within an agent system

|  |
| --- |
| # Copyright (c) 2025 Marco Fago  #  # This code is licensed under the MIT License.  # See the LICENSE file in the repository for the full license text.  import uuid  from typing import Dict, Any, Optional  from google.adk.agents import Agent  from google.adk.runners import InMemoryRunner  from google.adk.tools import FunctionTool  from google.genai import types  from google.adk.events import Event  # --- Define Tool Functions ---  # These functions simulate the actions of the specialist agents.  def booking\_handler(request: str) -> str:  """  Handles booking requests for flights and hotels.  Args:  request: The user's request for a booking.  Returns:  A confirmation message that the booking was handled.  """  print("-------------------------- Booking Handler Called ----------------------------")  return f"Booking action for '{request}' has been simulated."  def info\_handler(request: str) -> str:  """  Handles general information requests.  Args:  request: The user's question.  Returns:  A message indicating the information request was handled.  """  print("-------------------------- Info Handler Called ----------------------------")  return f"Information request for '{request}'. Result: Simulated information retrieval."  def unclear\_handler(request: str) -> str:  """Handles requests that couldn't be delegated."""  return f"Coordinator could not delegate request: '{request}'. Please clarify."  # --- Create Tools from Functions ---  booking\_tool = FunctionTool(booking\_handler)  info\_tool = FunctionTool(info\_handler)  # Define specialized sub-agents equipped with their respective tools  booking\_agent = Agent(  name="Booker",  model="gemini-2.0-flash",  description="A specialized agent that handles all flight  and hotel booking requests by calling the booking tool.",  tools=[booking\_tool]  )  info\_agent = Agent(  name="Info",  model="gemini-2.0-flash",  description="A specialized agent that provides general information  and answers user questions by calling the info tool.",  tools=[info\_tool]  )  # Define the parent agent with explicit delegation instructions  coordinator = Agent(  name="Coordinator",  model="gemini-2.0-flash",  instruction=(  "You are the main coordinator. Your only task is to analyze  incoming user requests "  "and delegate them to the appropriate specialist agent.  Do not try to answer the user directly.\n"  "- For any requests related to booking flights or hotels,  delegate to the 'Booker' agent.\n"  "- For all other general information questions, delegate to the 'Info' agent."  ),  description="A coordinator that routes user requests to the  correct specialist agent.",  # The presence of sub\_agents enables LLM-driven delegation (Auto-Flow) by default.  sub\_agents=[booking\_agent, info\_agent]  )  # --- Execution Logic ---  async  def run\_coordinator(runner: InMemoryRunner, request: str):  """Runs the coordinator agent with a given request and delegates."""  print(f"\n--- Running Coordinator with request: '{request}' ---")  final\_result = ""  try:  user\_id = "user\_123"  session\_id = str(uuid.uuid4())  await  runner.session\_service.create\_session(  app\_name=runner.app\_name, user\_id=user\_id, session\_id=session\_id  )  for event in runner.run(  user\_id=user\_id,  session\_id=session\_id,  new\_message=types.Content(  role='user',  parts=[types.Part(text=request)]  ),  ):  if event.is\_final\_response() and event.content:  # Try to get text directly from event.content  # to avoid iterating parts  if hasattr(event.content, 'text') and event.content.text:  final\_result = event.content.text  elif event.content.parts:  # Fallback: Iterate through parts and extract text (might trigger warning)  text\_parts = [part.text for part in event.content.parts if part.text]  final\_result = "".join(text\_parts)  # Assuming the loop should break after the final response  break  print(f"Coordinator Final Response: {final\_result}")  return final\_result  except Exception as e:  print(f"An error occurred while processing your request: {e}")  return f"An error occurred while processing your request: {e}"  async  def main():  """Main function to run the ADK example."""  print("--- Google ADK Routing Example (ADK Auto-Flow Style) ---")  print("Note: This requires Google ADK installed and authenticated.")  runner = InMemoryRunner(coordinator)  # Example Usage  result\_a = await run\_coordinator(runner, "Book me a hotel in Paris.")  print(f"Final Output A: {result\_a}")  result\_b = await run\_coordinator(runner, "What is the highest mountain in the world?")  print(f"Final Output B: {result\_b}")  result\_c = await run\_coordinator(runner, "Tell me a random fact.") # Should go to Info  print(f"Final Output C: {result\_c}")  result\_d = await run\_coordinator(runner, "Find flights to Tokyo next month.") # Should go to Booker  print(f"Final Output D: {result\_d}")  if \_\_name\_\_ == "\_\_main\_\_":  import nest\_asyncio  nest\_asyncio.apply()  await main() |

This script consists of a main Coordinator agent and two specialized sub\_agents: Booker and Info. Each specialized agent is equipped with a FunctionTool that wraps a Python function simulating an action. The booking\_handler function simulates handling flight and hotel bookings, while the info\_handler function simulates retrieving general information. The unclear\_handler is included as a fallback for requests the coordinator cannot delegate, although the current coordinator logic doesn't explicitly use it for delegation failure in the main run\_coordinator function.

The Coordinator agent's primary role, as defined in its instruction, is to analyze incoming user messages and delegate them to either the Booker or Info agent. This delegation is handled automatically by the ADK's Auto-Flow mechanism because the Coordinator has sub\_agents defined. The run\_coordinator function sets up an InMemoryRunner, creates a user and session ID, and then uses the runner to process the user's request through the coordinator agent. The runner.run method processes the request and yields events, and the code extracts the final response text from the event.content.

The main function demonstrates the system's usage by running the coordinator with different requests, showcasing how it delegates booking requests to the Booker and information requests to the Info agent.

**At a Glance**

**What**: Agentic systems must often respond to a wide variety of inputs and situations that cannot be handled by a single, linear process. A simple sequential workflow lacks the ability to make decisions based on context. Without a mechanism to choose the correct tool or sub-process for a specific task, the system remains rigid and non-adaptive. This limitation makes it difficult to build sophisticated applications that can manage the complexity and variability of real-world user requests.

**Why:** The Routing pattern provides a standardized solution by introducing conditional logic into an agent's operational framework. It enables the system to first analyze an incoming query to determine its intent or nature. Based on this analysis, the agent dynamically directs the flow of control to the most appropriate specialized tool, function, or sub-agent. This decision can be driven by various methods, including prompting LLMs, applying predefined rules, or using embedding-based semantic similarity. Ultimately, routing transforms a static, predetermined execution path into a flexible and context-aware workflow capable of selecting the best possible action.

**Rule of Thumb:** Use the Routing pattern when an agent must decide between multiple distinct workflows, tools, or sub-agents based on the user's input or the current state. It is essential for applications that need to triage or classify incoming requests to handle different types of tasks, such as a customer support bot distinguishing between sales inquiries, technical support, and account management questions.

**Visual Summary:**

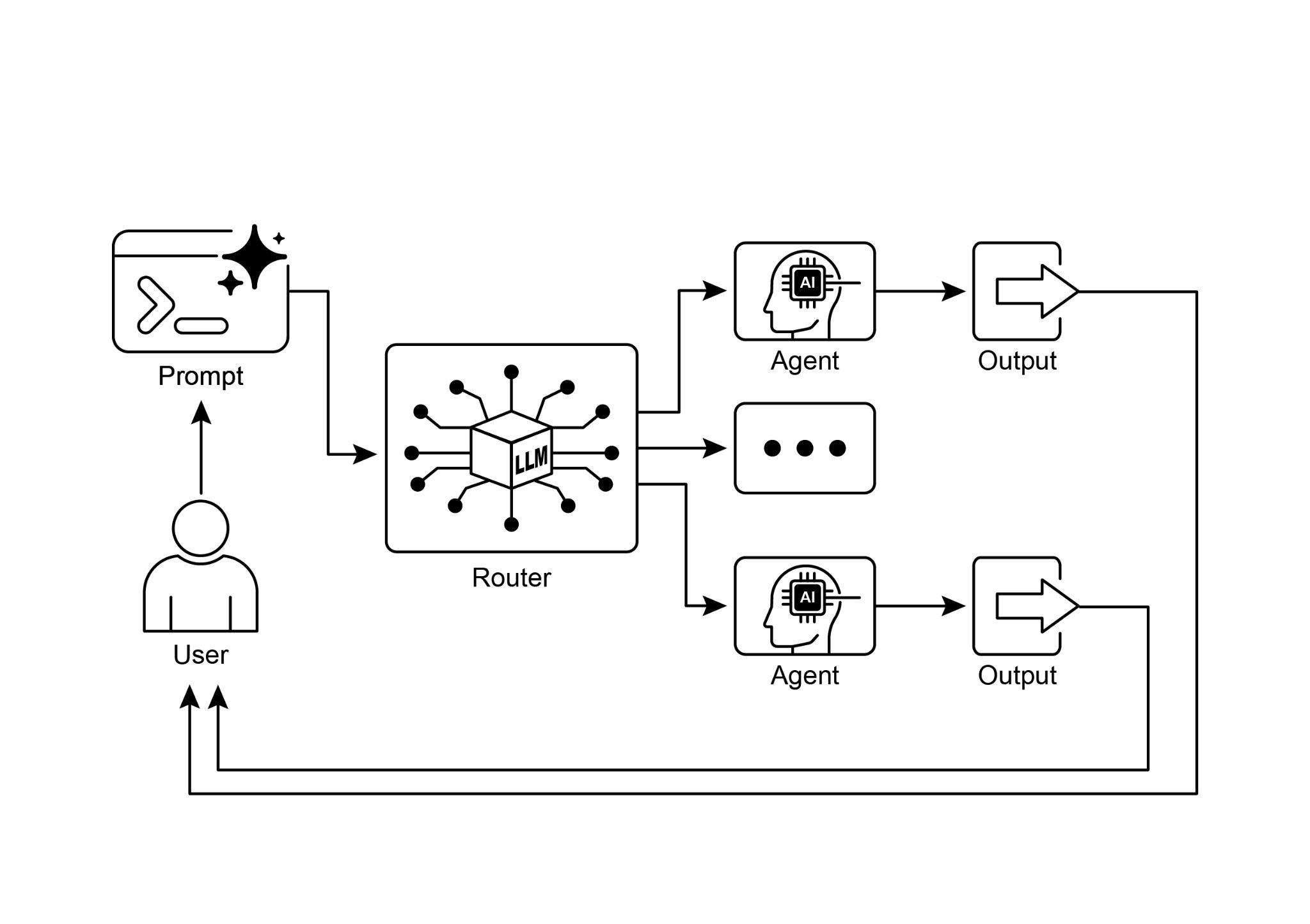


Fig.1: Router pattern, using an LLM as a Router

**Key Takeaways**

* Routing enables agents to make dynamic decisions about the next step in a workflow based on conditions.
* It allows agents to handle diverse inputs and adapt their behavior, moving beyond linear execution.
* Routing logic can be implemented using LLMs, rule-based systems, or embedding similarity.
* Frameworks like LangGraph and Google ADK provide structured ways to define and manage routing within agent workflows, albeit with different architectural approaches.

**Conclusion**

The Routing pattern is a critical step in building truly dynamic and responsive agentic systems. By implementing routing, we move beyond simple, linear execution flows and empower our agents to make intelligent decisions about how to process information, respond to user input, and utilize available tools or sub-agents.

We've seen how routing can be applied in various domains, from customer service chatbots to complex data processing pipelines. The ability to analyze input and conditionally direct the workflow is fundamental to creating agents that can handle the inherent variability of real-world tasks.

The code examples using LangChain and Google ADK demonstrate two different, yet effective, approaches to implementing routing. LangGraph's graph-based structure provides a visual and explicit way to define states and transitions, making it ideal for complex, multi-step workflows with intricate routing logic. Google ADK, on the other hand, often focuses on defining distinct capabilities (Tools) and relies on the framework's ability to route user requests to the appropriate tool handler, which can be simpler for agents with a well-defined set of discrete actions.

Mastering the Routing pattern is essential for building agents that can intelligently navigate different scenarios and provide tailored responses or actions based on context. It's a key component in creating versatile and robust agentic applications.

**References**

1. LangGraph Documentation: <https://www.langchain.com/>
2. Google Agent Developer Kit Documentation: <https://google.github.io/adk-docs/>

**第二章\_路由**

第二章：路由

**路由模式概述**

虽然通过提示链进行顺序处理是利用语言模型执行确定性线流的基础技术，但其适用性在需要自适应响应的场景中受到限制。现实世界中的智能体系统通常必须根据偶然因素（如环境状态、用户输入或先前操作的结果）在多个潜在行动之间进行仲裁。这种动态决策能力（它控制着对不同专门功能、工具或子流程的控制流）是通过一种称为路由的机制实现的。

路由将条件逻辑引入到代理的操作框架中，使代理能够从固定的执行路径转变为一种动态评估特定标准，从而从一组可能的后续行动中进行选择的模式。这使得系统行为更加灵活且具有上下文感知能力。

例如，一个专为处理客户咨询而设计的智能体，当配备路由功能时，它可以先对收到的查询进行分类，以确定用户的意图。基于这种分类，它可以将查询引导至专门的智能体进行直接问答、引导至用于检索账户信息的数据库工具，或者引导至处理复杂问题的升级流程，而不是默认采用单一的、预先确定的响应路径。因此，使用路由功能的更复杂智能体可以：

1. 分析用户的查询。
2. **根据查询的**意图*路由查询*：

* 如果意图是“查询订单状态”，则路由到与订单数据库交互的子代理或工具链。
* 如果意图是“产品信息”，则路由到搜索产品曲库的子代理或链。
* 如果意图是“技术支持”，则路由到另一个链，该链可访问故障排除指南或升级到人工处理。
* 如果意图不明确，则路由到澄清子代理或提示链。

路由模式的核心组件是一种执行评估并引导流程的机制。这种机制可以通过多种方式实现：

* **基于大语言模型（LLM）的路由：**语言模型本身可以被提示分析输入，并输出一个特定的标识符或指令，以指示下一步或目的地。例如，一个提示可能要求大语言模型“分析以下用户查询，并仅输出类别：'订单状态'、'产品信息'、'技术支持'或'其他'”。然后，代理系统读取此输出，并相应地引导工作流程。
* **基于嵌入的路由：**输入查询可以转换为向量嵌入（见RAG，第14章）。然后将此嵌入与代表不同路由或能力的嵌入进行比较。查询将被路由到嵌入最相似的路由。这对于语义路由很有用，在语义路由中，决策是基于输入的含义，而不仅仅是关键词。
* **基于规则的路由：**这涉及使用基于从输入中提取的关键字、模式或结构化数据的预设规则或逻辑（例如，if-else语句、switch case语句）。与基于大语言模型（LLM）的路由相比，这种方法可能更快且更具确定性，但在处理细微差别或新颖输入时灵活性较差。
* **基于机器学习模型的路由**：它采用判别模型，如分类器，该模型经过专门训练，使用少量标记数据语料库来执行路由任务。虽然它与基于嵌入的方法在概念上有相似之处，但其关键特征是有监督的微调过程，该过程调整模型的参数以创建专门的路由功能。这种技术与基于大语言模型（LLM）的路由不同，因为决策组件不是在推理时执行提示的生成模型。相反，路由逻辑被编码在微调模型的学习权重中。虽然大语言模型可能在预处理步骤中用于生成合成数据以扩充训练集，但它们不参与实时路由决策本身。

路由机制可以在代理操作周期内的多个节点上实现。它们可以在一开始就用于对主要任务进行分类，在处理链的中间点用于确定后续行动，或者在子程序中用于从给定集合中选择最合适的工具。

像LangChain、LangGraph和谷歌的Agent Developer Kit（ADK）这样的计算框架，为定义和管理此类条件逻辑提供了明确的结构。凭借其基于状态的图架构，LangGraph特别适合复杂的路由场景，在这些场景中，决策取决于整个系统的累积状态。同样，谷歌的ADK为构建代理的能力和交互模型提供了基础组件，这些组件是实现路由逻辑的基础。在这些框架提供的执行环境中，开发人员定义可能的操作路径，以及决定计算图中节点之间转换的函数或基于模型的评估。

路由的实现使系统能够超越确定性的顺序处理。它有助于开发更具适应性的执行流程，这些流程可以动态且适当地响应更广泛的输入和状态变化。

**实际应用与用例**

路由模式是自适应智能体系统设计中的关键控制机制，使系统能够根据可变输入和内部状态动态改变其执行路径。通过提供必要的条件逻辑层，其效用跨越多个领域。

在人机交互中，例如与虚拟助手或AI驱动的导师交互时，路由被用于解读用户意图。对自然语言查询的初步分析会确定最合适的后续行动，无论是调用特定的信息检索工具、升级到人工操作员，还是根据用户表现选择课程中的下一个模块。这使得系统能够超越线性对话流程，并进行上下文相关的响应。

在自动化数据和文档处理流程中，路由起着分类和分发的作用。系统会根据内容、元数据或格式对传入的数据（如电子邮件、支持工单或 API 负载）进行分析。然后，系统会将每个项目导向相应的工作流程，如潜在客户录入流程、针对 JSON 或 CSV 格式的特定数据转换功能，或紧急问题升级路径。

在涉及多个专业工具或智能体的复杂系统中，路由充当着高级调度器的角色。一个由负责搜索、总结和分析信息的不同智能体组成的研究系统，会使用路由器根据当前目标将任务分配给最合适的智能体。同样，AI编码助手在将代码片段传递给正确的专业工具之前，会使用路由来识别编程语言和用户意图（如调试、解释或翻译）。

最终，路由提供了逻辑仲裁的能力，这对于创建功能多样且具有上下文感知能力的系统至关重要。它将代理从预定义序列的静态执行者转变为一个动态系统，该系统能够在不断变化的条件下就完成任务的最有效方法做出决策。

**实践代码示例（LangChain）**

在代码中实现路由涉及定义可能的路径以及决定选择哪条路径的逻辑。像LangChain和LangGraph这样的框架为此提供了特定的组件和结构。LangGraph基于状态的图结构在可视化和实现路由逻辑方面尤其直观。

此代码展示了一个使用LangChain和谷歌生成式AI的简单类代理系统。它设置了一个“协调器”，根据用户请求的意图（预订、信息或意图不明确）将用户请求路由到不同的模拟“子代理”处理程序。该系统使用语言模型对请求进行分类，然后将其委派给适当的处理函数，模拟多代理架构中常见的基本委派模式。

首先，确保你已经安装了必要的库：

|  |
| --- |
| pip install langchain langgraph google-cloud-aiplatform langchain-google-genai google-adk deprecated pydantic |

你还需要使用你选择的语言模型（例如OpenAI、Google Gemini、Anthropic）的API密钥来设置你的环境。

|  |
| --- |
| #版权所有 (c) 2025 马尔科·法戈  # https://www.linkedin.com/in/marco-fago/  #  # 本代码遵循MIT许可证。  #请查看仓库中的 LICENSE 文件以获取完整的许可文本。  来自 langchain\_google\_genai 的 ChatGoogleGenerativeAI  from langchain\_core.prompts import ChatPromptTemplate  从 langchain\_core.output\_parsers 导入 StrOutputParser  来自 langchain\_core.runnables 的 RunnablePassthrough、RunnableBranch  # --- 配置 ---  # 确保已设置 API 密钥环境变量（例如，GOOGLE\_API\_KEY）  try:  llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)  print(f"语言模型已初始化：{llm.model}")  except Exception as e:  print(f"初始化语言模型时出错: {e}")  llm = None  # --- 定义模拟子代理处理程序（相当于ADK子代理） ---  def booking\_handler(request: str) -> str:  """模拟处理请求的预订代理。"""  print("\n--- 委托给预订处理程序 ---")  return f"预订处理程序已处理请求：'{request}'。结果：模拟预订操作。"  def info\_handler(request: str) -> str:  """模拟信息代理处理请求的过程。"""  print("\n--- 委托给信息处理程序 ---")  return f"信息处理程序已处理请求：'{request}'。结果：模拟信息检索。"  def unclear\_handler(request: str) -> str:  """处理无法委派的请求。"""  print("\n--- 处理不明确的请求 ---")  return f"协调器无法委派请求：'{request}'。请澄清。"  # --- 定义协调器路由器链（相当于ADK协调器的指令） ---  # 此链决定要委托给哪个处理程序。  coordinator\_router\_prompt = ChatPromptTemplate.from\_messages([  （"系统"，"""分析用户请求并确定应由哪个专家处理程序来处理它。  - 如果请求与预订航班或酒店相关，  输出 'booker'。  - 对于所有其他一般信息问题，请输出'info'。  - 如果请求不明确或不属于任何一类，  输出'不清晰'。  仅输出一个单词：'booker'、'info'或'unclear'。  ("用户", "{请求}")  ])  如果大语言模型存在：  coordinator\_router\_chain = coordinator\_router\_prompt | llm | StrOutputParser()  # --- 定义委托逻辑（相当于基于子代理的 ADK 自动流） ---  # 使用 RunnableBranch 根据路由器链的输出进行路由。  # 定义RunnableBranch的分支  分支 = {  "booker": RunnablePassthrough.assign(output=lambda x: booking\_handler(x['request']['request'])),  "info": RunnablePassthrough.assign(output=lambda x: info\_handler(x['request']['request'])),  "unclear": RunnablePassthrough.assign(output=lambda x: unclear\_handler(x['request']['request'])),  }  # 创建RunnableBranch。它接收路由器链的输出  #并将原始输入（'请求'）路由到相应的处理程序。  delegation\_branch = RunnableBranch(  (lambda x: x['decision'].strip() == 'booker', branches["booker"]), # Added.strip()  (lambda x: x['decision'].strip() == 'info', branches["info"]), # Added.strip()  branches["unclear"] # 'unclear'或任何其他输出的默认分支  )  # 将路由器链和委托分支合并为一个可运行的单元  #路由器链的输出（“决策”）与原始输入（“请求”）一起传递  # 到代表团分支。  coordinator\_agent = {  "决策": coordinator\_router\_chain,  "request": RunnablePassthrough()  } | delegation\_branch | (lambda x: x['output']) # 提取最终输出  # --- 示例用法 ---  def main():  if not llm:  print("\n由于大语言模型（LLM）初始化失败，跳过执行。")  返回  print("--- 正在处理预订请求 ---")  request\_a = "帮我预订一张去伦敦的机票。"  result\_a = coordinator\_agent.invoke({"request": request\_a})  print(f"最终结果A: {result\_a}")  print("\n--- 正在运行信息请求 ---")  request\_b = "意大利的首都是哪里？"  result\_b = coordinator\_agent.invoke({"request": request\_b})  print(f"最终结果B: {result\_b}")  print("\n--- 正在处理不明确的请求 ---")  request\_c = "给我讲讲量子物理学。"  result\_c = coordinator\_agent.invoke({"request": request\_c})  print(f"最终结果C: {result\_c}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

如前所述，这段Python代码使用LangChain库和谷歌的生成式AI模型（具体为gemini-2.5-flash）构建了一个简单的类代理系统。详细来说，它定义了三个模拟的子代理处理程序：booking\_handler、info\_handler和unclear\_handler，每个处理程序都旨在处理特定类型的请求。

核心组件是coordinator\_router\_chain，它利用ChatPromptTemplate指示语言模型将传入的用户请求分类为以下三个类别之一：'booker'、'info'或'unclear'。此路由链的输出随后由RunnableBranch使用，将原始请求委派给相应的处理函数。RunnableBranch检查语言模型的决策，并将请求数据导向booking\_handler、info\_handler或unclear\_handler。coordinator\_agent将这些组件组合在一起，首先对请求进行路由以做出决策，然后将请求传递给选定的处理函数。最终输出从处理函数的响应中提取。

主函数通过三个示例请求展示了系统的使用方法，展示了不同的输入如何被模拟代理路由和处理。包含语言模型初始化的错误处理，以确保系统的健壮性。代码结构模仿了一个基本的多智能体框架，其中中央协调器根据意图将任务委派给专门的代理。

**实践代码示例（谷歌ADK）**

代理开发套件（ADK）是用于设计智能体系统的框架，它提供了一个结构化环境，用于定义智能体的能力和行为。与基于显式计算图的架构不同，ADK范式中的路由通常通过定义一组离散的“工具”来实现，这些工具代表智能体的功能。框架的内部逻辑负责管理响应于用户查询的适当工具的选择，该逻辑利用底层模型将用户意图与正确的功能处理程序进行匹配。

这段Python代码展示了一个使用谷歌代理开发工具包（ADK）库的ADK应用示例。它设置了一个“协调器”代理，根据既定指令将用户请求路由到专门的子代理（“预订员”负责预订，“信息员”负责提供一般信息）。然后，子代理使用特定工具模拟处理请求，展示了代理系统内的基本委托模式

|  |
| --- |
| #版权所有 (c) 2025 马尔科·法戈  #  # 本代码遵循MIT许可证。  #请查看仓库中的 LICENSE 文件以获取完整的许可文本。  导入uuid  从 typing 导入 Dict、Any、Optional  from google.adk.agents import Agent  from google.adk.runners import InMemoryRunner  from google.adk.tools import FunctionTool  从google.genai导入types  from google.adk.events import Event  # --- 定义工具函数 ---  # 这些函数模拟了专家代理的行为。  def booking\_handler(request: str) -> str:  """  处理航班和酒店的预订请求。  参数：  请求：用户的预订请求。  返回值：  一条确认消息，表明预订已处理。  """  print("-------------------------- 预订处理程序已调用 ----------------------------")  return f"已模拟针对 '{request}' 的预订操作。"  def info\_handler(request: str) -> str:  """  处理一般信息请求。  参数：  请求：用户的问题。  返回值：  一条指示信息请求已被处理的消息。  """  print("-------------------------- Info Handler Called ----------------------------")  return f"信息请求为 '{request}'。结果：模拟信息检索。"  def unclear\_handler(request: str) -> str:  """处理无法委派的请求。"""  return f"协调器无法委派请求：'{request}'。请澄清。"  # --- 从函数创建工具 ---  booking\_tool = FunctionTool(booking\_handler)  info\_tool = FunctionTool(info\_handler)  # 定义配备各自工具的专业子智能体  booking\_agent = Agent(  name="布克",  model="gemini-2.0-flash",  description="专门处理所有航班的代理  并通过调用预订工具进行酒店预订请求。  tools=[booking\_tool]  )  info\_agent = Agent(  name="Info",  model="gemini-2.0-flash",  description="提供一般信息的专业代理  并通过调用信息工具来回答用户问题。  tools=[info\_tool]  )  # 定义具有明确委派指令的父代理  coordinator = Agent(  name="协调员",  model="gemini-2.0-flash",  instruction=(  你是主要协调人。你的唯一任务是分析  传入的用户请求“  并将其委派给合适的专业代理。  不要试图直接回答用户。  "- 对于任何与预订航班或酒店相关的请求，  委托给“布克”代理。  "- 对于所有其他一般信息问题，委托给'信息'代理处理。"  ),  描述="将用户请求路由到  正确的专家代理。  # 子代理的存在默认启用了大语言模型驱动的委派（自动流程）。  sub\_agents=[booking\_agent, info\_agent]  )  # --- 执行逻辑 ---  异步  def run\_coordinator(runner: InMemoryRunner, request: str):  """使用给定的请求和委托运行协调器代理。"""  print(f"\n--- 使用请求 '{request}' 运行协调器 ---")  final\_result = ""  try:  user\_id = "user\_123"  session\_id = str(uuid.uuid4())  等待  runner.session\_service.create\_session(  app\_name=runner.app\_name, user\_id=用户ID, session\_id=会话ID  )  for event in runner.run(  user\_id=用户ID,  session\_id=会话ID,  new\_message=types.Content(  role='用户',  parts=[types.Part(text=请求)]  ),  ):  if event.is\_final\_response() and event.content:  # 尝试直接从 event.content 获取文本  # 避免迭代部分  if hasattr(event.content, 'text') and event.content.text:  final\_result = event.content.text  elif event.content.parts:  # 回退：遍历各部分并提取文本（可能会触发警告）  text\_parts = [part.text for part in event.content.parts if part.text]  final\_result = "".join(text\_parts)  # 假设循环应在最终响应后中断  中断  print(f"协调器最终响应: {final\_result}")  返回最终结果  except Exception as e:  print(f"处理您的请求时发生错误: {e}")  return f"处理您的请求时发生错误：{e}"  异步  def main():  """运行ADK示例的主函数。"""  print("--- Google ADK路由示例（ADK自动流风格） ---")  print("注意：这需要安装并认证Google ADK。")  runner = InMemoryRunner(coordinator)  # 示例用法  result\_a = await run\_coordinator(runner, "帮我在巴黎预订一家酒店。")  print(f"最终输出A: {result\_a}")  result\_b = await run\_coordinator(runner, "世界上最高的山是什么？")  print(f"最终输出B: {result\_b}")  result\_c = await run\_coordinator(runner, "给我讲一个随机的事实。") # 应该转到信息  print(f"最终输出C: {result\_c}")  result\_d = await run\_coordinator(runner, "查找下个月飞往东京的航班。") # 应转至预订员  print(f"最终输出 D: {result\_d}")  if \_\_name\_\_ == "\_\_main\_\_":  import nest\_asyncio  nest\_asyncio.apply()  await main() |

此脚本由一个主协调器代理和两个专门的子代理组成：预订器（Booker）和信息器（Info）。每个专门的代理都配备了一个功能工具（FunctionTool），该工具封装了一个模拟操作的Python函数。预订处理函数（booking\_handler）模拟处理航班和酒店预订，而信息处理函数（info\_handler）模拟检索一般信息。不明确处理函数（unclear\_handler）作为协调器无法委派的请求的备用处理方式，尽管当前协调器逻辑在主运行协调器函数（run\_coordinator）中并未明确将其用于委派失败的情况。

根据其指令的定义，协调器代理的主要职责是分析传入的用户消息，并将其委派给预订员或信息代理。由于协调器定义了子代理，因此这种委派由ADK的自动流机制自动处理。run\_coordinator函数设置一个内存内运行器，创建用户和会话ID，然后使用该运行器通过协调器代理处理用户的请求。运行器的run方法处理请求并生成事件，代码从事件内容中提取最终响应文本。

主函数通过使用不同的请求运行协调器来展示系统的使用方式，展示了它如何将预订请求委托给预订员，将信息请求委托给信息代理。

**概览**

**什么**：能动系统往往必须对各种各样的输入和情况做出响应，而这些是单一的线无法处理的。简单的顺序工作流程缺乏根据上下文做出决策的能力。如果没有一种机制来为特定任务选择正确的工具或子流程，系统就会保持僵化且缺乏适应性。这一局限性使得构建能够处理现实世界用户请求的复杂性和可的复杂应用程序变得困难。

**原因：**路由模式通过将条件逻辑引入代理的操作框架，提供了一种标准化的解决方案。它使系统能够首先分析传入的查询，以确定其意图或性质。基于此分析，代理动态地将控制流导向最合适的专业工具、功能或子代理。这一决策可以通过多种方法来驱动，包括提示大语言模型、应用预设规则或使用基于嵌入的语义相似度。最终，路由将静态的、预先确定的执行路径转变为能够选择最佳可能行动的灵活且具有上下文感知能力的工作流程。

**经验法则：**当代理必须根据用户输入或当前状态在多个不同的工作流、工具或子代理之间做出决策时，请使用路由模式。对于需要对传入请求进行分类或归类以处理不同类型任务的应用程序来说，这一点至关重要，例如客户支持机器人区分销售咨询、技术支持和账户管理问题。

**可视化总结：**

图1：路由模式，使用大语言模型作为路由器

**要点总结**

* 路由使座席能够根据条件对工作流中的下一步做出动态决策。
* 它使智能体能够处理多样化的输入并调整其行为，超越线性执行。
* 路由逻辑可以使用大语言模型（LLMs）、基于规则的系统或嵌入相似度来实现。
* 像LangGraph和谷歌ADK这样的框架提供了结构化的方法来定义和管理代理工作流中的路由，尽管采用了不同的架构方法。

**结论**

路由模式是构建真正动态且响应式的智能体系统的关键步骤。通过实施路由，我们超越了简单的线性执行流程，使智能体能够就如何处理信息、响应用户输入以及利用可用工具或子智能体做出明智决策。

我们已经看到路由如何应用于各个领域，从客户服务聊天机器人到复杂的数据处理管道。分析输入并根据条件引导工作流程的能力，是创建能够处理现实世界任务中固有可的智能体的基础。

使用LangChain和Google ADK的代码示例展示了两种不同但有效的路由实现方法。LangGraph基于图的结构提供了一种可视化且明确的方式来定义状态和转换，使其非常适合具有复杂路由逻辑的复杂多步骤工作流。另一方面，Google ADK通常侧重于定义不同的能力（工具），并依赖于框架将用户请求路由到适当工具处理程序的能力，对于具有明确定义的离散操作集的代理来说，这种方法可能更简单。

掌握路由模式对于构建能够智能应对不同场景，并根据上下文提供定制化响应或行动的智能体至关重要。它是创建多功能、强大的智能体应用的关键组成部分。

**参考文献**

1. LangGraph文档：<https://www.langchain.com/>
2. Google Agent Developer Kit 文档：<https://google.github.io/adk-docs/>