**Chapter 7\_ Multi-Agent Collaboration**

Chapter 7: Multi-Agent Collaboration

While a monolithic agent architecture can be effective for well-defined problems, its capabilities are often constrained when faced with complex, multi-domain tasks. The Multi-Agent Collaboration pattern addresses these limitations by structuring a system as a cooperative ensemble of distinct, specialized agents. This approach is predicated on the principle of task decomposition, where a high-level objective is broken down into discrete sub-problems. Each sub-problem is then assigned to an agent possessing the specific tools, data access, or reasoning capabilities best suited for that task.

For example, a complex research query might be decomposed and assigned to a Research Agent for information retrieval, a Data Analysis Agent for statistical processing, and a Synthesis Agent for generating the final report. The efficacy of such a system is not merely due to the division of labor but is critically dependent on the mechanisms for inter-agent communication. This requires a standardized communication protocol and a shared ontology, allowing agents to exchange data, delegate sub-tasks, and coordinate their actions to ensure the final output is coherent.

This distributed architecture offers several advantages, including enhanced modularity, scalability, and robustness, as the failure of a single agent does not necessarily cause a total system failure. The collaboration allows for a synergistic outcome where the collective performance of the multi-agent system surpasses the potential capabilities of any single agent within the ensemble.

**Multi-Agent Collaboration Pattern Overview**

The Multi-Agent Collaboration pattern involves designing systems where multiple independent or semi-independent agents work together to achieve a common goal. Each agent typically has a defined role, specific goals aligned with the overall objective, and potentially access to different tools or knowledge bases. The power of this pattern lies in the interaction and synergy between these agents.

Collaboration can take various forms:

* **Sequential Handoffs:** One agent completes a task and passes its output to another agent for the next step in a pipeline (similar to the Planning pattern, but explicitly involving different agents).
* **Parallel Processing:** Multiple agents work on different parts of a problem simultaneously, and their results are later combined.
* **Debate and Consensus:** Multi-Agent Collaboration where Agents with varied perspectives and information sources engage in discussions to evaluate options, ultimately reaching a consensus or a more informed decision.
* **Hierarchical Structures:** A manager agent might delegate tasks to worker agents dynamically based on their tool access or plugin capabilities and synthesize their results. Each agent can also handle relevant groups of tools, rather than a single agent handling all the tools.
* **Expert Teams:** Agents with specialized knowledge in different domains (e.g., a researcher, a writer, an editor) collaborate to produce a complex output.
* **Critic-Reviewer:** Agents create initial outputs such as plans, drafts, or answers. A second group of agents then critically assesses this output for adherence to policies, security, compliance, correctness, quality, and alignment with organizational objectives. The original creator or a final agent revises the output based on this feedback. This pattern is particularly effective for code generation, research writing, logic checking, and ensuring ethical alignment. The advantages of this approach include increased robustness, improved quality, and a reduced likelihood of hallucinations or errors.

A multi-agent system (see Fig.1) fundamentally comprises the delineation of agent roles and responsibilities, the establishment of communication channels through which agents exchange information, and the formulation of a task flow or interaction protocol that directs their collaborative endeavors.

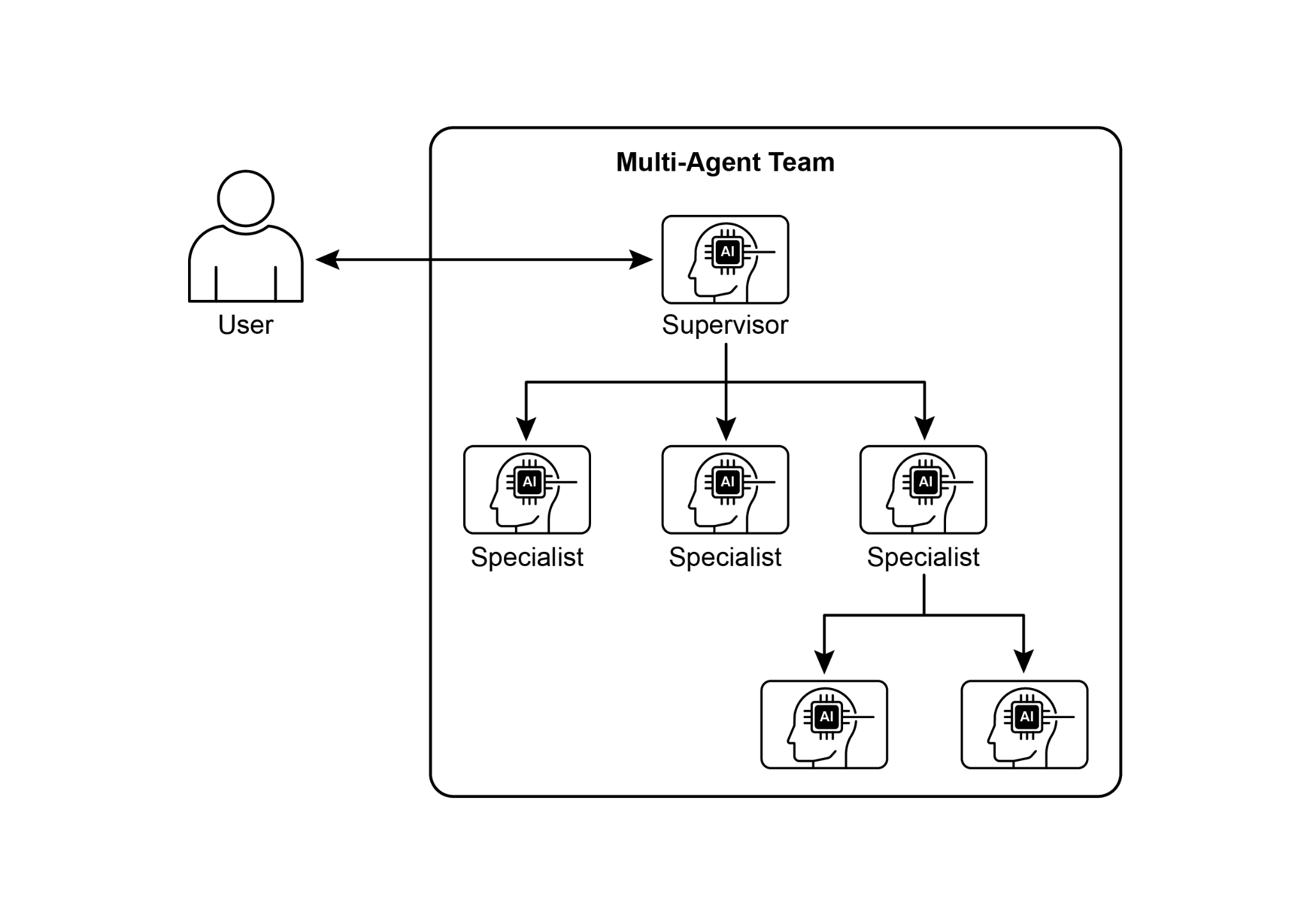


Fig.1: Example of multi-agent system

Frameworks such as Crew AI and Google ADK are engineered to facilitate this paradigm by providing structures for the specification of agents, tasks, and their interactive procedures. This approach is particularly effective for challenges necessitating a variety of specialized knowledge, encompassing multiple discrete phases, or leveraging the advantages of concurrent processing and the corroboration of information across agents.

**Practical Applications & Use Cases**

Multi-Agent Collaboration is a powerful pattern applicable across numerous domains:

* **Complex Research and Analysis:** A team of agents could collaborate on a research project. One agent might specialize in searching academic databases, another in summarizing findings, a third in identifying trends, and a fourth in synthesizing the information into a report. This mirrors how a human research team might operate.
* **Software Development:** Imagine agents collaborating on building software. One agent could be a requirements analyst, another a code generator, a third a tester, and a fourth a documentation writer. They could pass outputs between each other to build and verify components.
* **Creative Content Generation:** Creating a marketing campaign could involve a market research agent, a copywriter agent, a graphic design agent (using image generation tools), and a social media scheduling agent, all working together.
* **Financial Analysis:** A multi-agent system could analyze financial markets. Agents might specialize in fetching stock data, analyzing news sentiment, performing technical analysis, and generating investment recommendations.
* **Customer Support Escalation:** A front-line support agent could handle initial queries, escalating complex issues to a specialist agent (e.g., a technical expert or a billing specialist) when needed, demonstrating a sequential handoff based on problem complexity.
* **Supply Chain Optimization:** Agents could represent different nodes in a supply chain (suppliers, manufacturers, distributors) and collaborate to optimize inventory levels, logistics, and scheduling in response to changing demand or disruptions.
* **Network Analysis & Remediation**: Autonomous operations benefit greatly from an agentic architecture, particularly in failure pinpointing. Multiple agents can collaborate to triage and remediate issues, suggesting optimal actions. These agents can also integrate with traditional machine learning models and tooling, leveraging existing systems while simultaneously offering the advantages of Generative AI.

The capacity to delineate specialized agents and meticulously orchestrate their interrelationships empowers developers to construct systems exhibiting enhanced modularity, scalability, and the ability to address complexities that would prove insurmountable for a singular, integrated agent.

**Multi-Agent Collaboration: Exploring Interrelationships and Communication Structures**

Understanding the intricate ways in which agents interact and communicate is fundamental to designing effective multi-agent systems. As depicted in Fig. 2, a spectrum of interrelationship and communication models exists, ranging from the simplest single-agent scenario to complex, custom-designed collaborative frameworks. Each model presents unique advantages and challenges, influencing the overall efficiency, robustness, and adaptability of the multi-agent system.

**1. Single Agent:** At the most basic level, a "Single Agent" operates autonomously without direct interaction or communication with other entities. While this model is straightforward to implement and manage, its capabilities are inherently limited by the individual agent's scope and resources. It is suitable for tasks that are decomposable into independent sub-problems, each solvable by a single, self-sufficient agent.

**2. Network:** The "Network" model represents a significant step towards collaboration, where multiple agents interact directly with each other in a decentralized fashion. Communication typically occurs peer-to-peer, allowing for the sharing of information, resources, and even tasks. This model fosters resilience, as the failure of one agent does not necessarily cripple the entire system. However, managing communication overhead and ensuring coherent decision-making in a large, unstructured network can be challenging.

**3. Supervisor:** In the "Supervisor" model, a dedicated agent, the "supervisor," oversees and coordinates the activities of a group of subordinate agents. The supervisor acts as a central hub for communication, task allocation, and conflict resolution. This hierarchical structure offers clear lines of authority and can simplify management and control. However, it introduces a single point of failure (the supervisor) and can become a bottleneck if the supervisor is overwhelmed by a large number of subordinates or complex tasks.

**4. Supervisor as a Tool:** This model is a nuanced extension of the "Supervisor" concept, where the supervisor's role is less about direct command and control and more about providing resources, guidance, or analytical support to other agents. The supervisor might offer tools, data, or computational services that enable other agents to perform their tasks more effectively, without necessarily dictating their every action. This approach aims to leverage the supervisor's capabilities without imposing rigid top-down control.

**5. Hierarchical:** The "Hierarchical" model expands upon the supervisor concept to create a multi-layered organizational structure. This involves multiple levels of supervisors, with higher-level supervisors overseeing lower-level ones, and ultimately, a collection of operational agents at the lowest tier. This structure is well-suited for complex problems that can be decomposed into sub-problems, each managed by a specific layer of the hierarchy. It provides a structured approach to scalability and complexity management, allowing for distributed decision-making within defined boundaries.

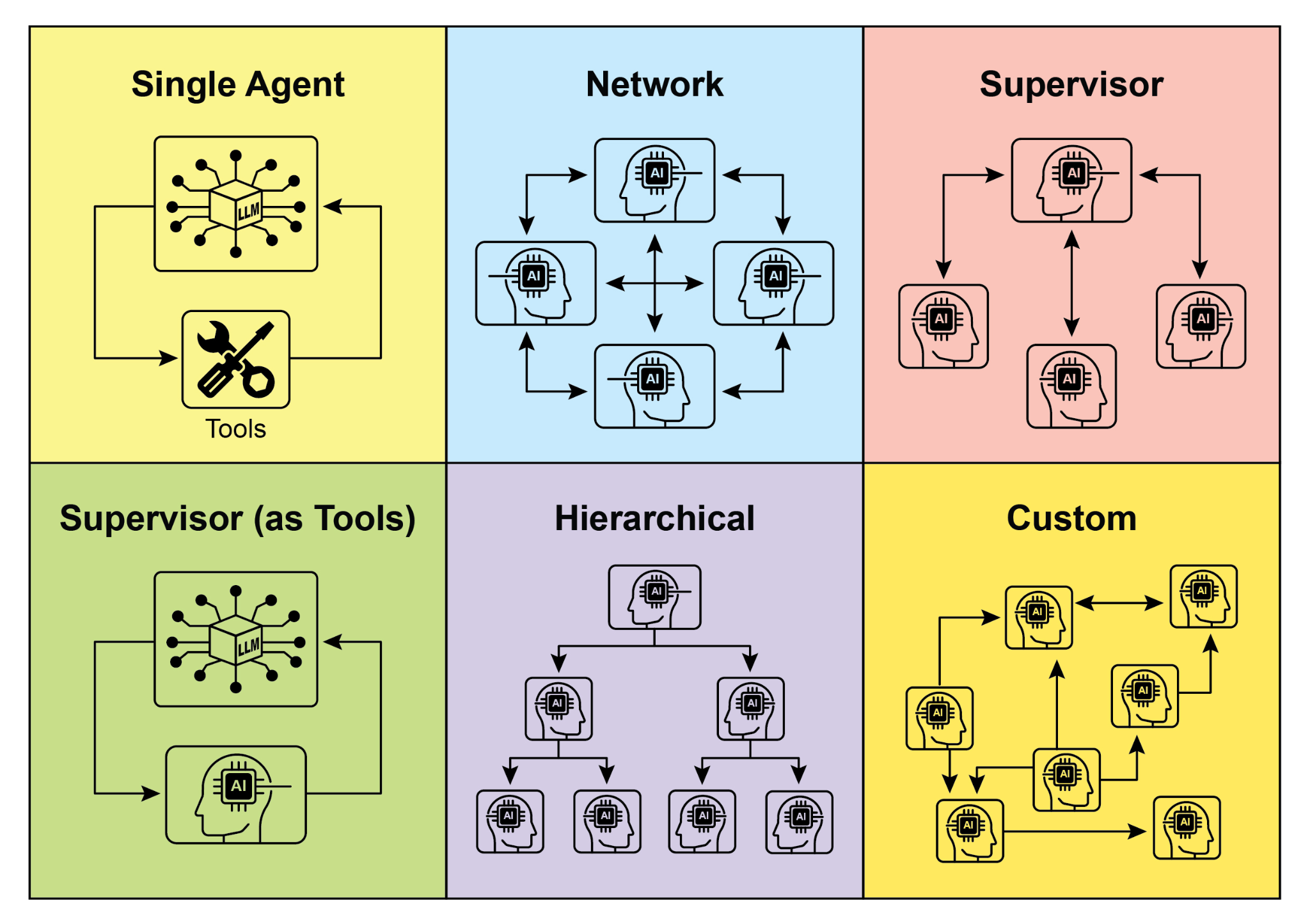


Fig. 2: Agents communicate and interact in various ways.

**6. Custom:** The "Custom" model represents the ultimate flexibility in multi-agent system design. It allows for the creation of unique interrelationship and communication structures tailored precisely to the specific requirements of a given problem or application. This can involve hybrid approaches that combine elements from the previously mentioned models, or entirely novel designs that emerge from the unique constraints and opportunities of the environment. Custom models often arise from the need to optimize for specific performance metrics, handle highly dynamic environments, or incorporate domain-specific knowledge into the system's architecture. Designing and implementing custom models typically requires a deep understanding of multi-agent systems principles and careful consideration of communication protocols, coordination mechanisms, and emergent behaviors.

In summary, the choice of interrelationship and communication model for a multi-agent system is a critical design decision. Each model offers distinct advantages and disadvantages, and the optimal choice depends on factors such as the complexity of the task, the number of agents, the desired level of autonomy, the need for robustness, and the acceptable communication overhead. Future advancements in multi-agent systems will likely continue to explore and refine these models, as well as develop new paradigms for collaborative intelligence.

**Hands-On code (Crew AI)**

This Python code defines an AI-powered crew using the CrewAI framework to generate a blog post about AI trends. It starts by setting up the environment, loading API keys from a .env file. The core of the application involves defining two agents: a researcher to find and summarize AI trends, and a writer to create a blog post based on the research.

Two tasks are defined accordingly: one for researching the trends and another for writing the blog post, with the writing task depending on the output of the research task. These agents and tasks are then assembled into a Crew, specifying a sequential process where tasks are executed in order. The Crew is initialized with the agents, tasks, and a language model (specifically the "gemini-2.0-flash" model). The main function executes this crew using the kickoff() method, orchestrating the collaboration between the agents to produce the desired output. Finally, the code prints the final result of the crew's execution, which is the generated blog post.

|  |
| --- |
| import os  from dotenv import load\_dotenv  from crewai import Agent, Task, Crew, Process  from langchain\_google\_genai import ChatGoogleGenerativeAI  def setup\_environment():  """Loads environment variables and checks for the required API key."""  load\_dotenv()  if not os.getenv("GOOGLE\_API\_KEY"):  raise ValueError("GOOGLE\_API\_KEY not found. Please set it in your .env file.")  def main():  """  Initializes and runs the AI crew for content creation using the latest Gemini model.  """  setup\_environment()  # Define the language model to use.  # Updated to a model from the Gemini 2.0 series for better performance and features.  # For cutting-edge (preview) capabilities, you could use "gemini-2.5-flash".  llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")  # Define Agents with specific roles and goals  researcher = Agent(  role='Senior Research Analyst',  goal='Find and summarize the latest trends in AI.',  backstory="You are an experienced research analyst with a knack for identifying key trends and synthesizing information.",  verbose=True,  allow\_delegation=False,  )  writer = Agent(  role='Technical Content Writer',  goal='Write a clear and engaging blog post based on research findings.',  backstory="You are a skilled writer who can translate complex technical topics into accessible content.",  verbose=True,  allow\_delegation=False,  )  # Define Tasks for the agents  research\_task = Task(  description="Research the top 3 emerging trends in Artificial Intelligence in 2024-2025. Focus on practical applications and potential impact.",  expected\_output="A detailed summary of the top 3 AI trends, including key points and sources.",  agent=researcher,  )  writing\_task = Task(  description="Write a 500-word blog post based on the research findings. The post should be engaging and easy for a general audience to understand.",  expected\_output="A complete 500-word blog post about the latest AI trends.",  agent=writer,  context=[research\_task],  )  # Create the Crew  blog\_creation\_crew = Crew(  agents=[researcher, writer],  tasks=[research\_task, writing\_task],  process=Process.sequential,  llm=llm,  verbose=2 # Set verbosity for detailed crew execution logs  )  # Execute the Crew  print("## Running the blog creation crew with Gemini 2.0 Flash... ##")  try:  result = blog\_creation\_crew.kickoff()  print("\n------------------\n")  print("## Crew Final Output ##")  print(result)  except Exception as e:  print(f"\nAn unexpected error occurred: {e}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

We will now delve into further examples within the Google ADK framework, with particular emphasis on hierarchical, parallel, and sequential coordination paradigms, alongside the implementation of an agent as an operational instrument.

**Hands-on Code (Google ADK)**

The following code example demonstrates the establishment of a hierarchical agent structure within the Google ADK through the creation of a parent-child relationship. The code defines two types of agents: LlmAgent and a custom TaskExecutor agent derived from BaseAgent. The TaskExecutor is designed for specific, non-LLM tasks and in this example, it simply yields a "Task finished successfully" event. An LlmAgent named greeter is initialized with a specified model and instruction to act as a friendly greeter. The custom TaskExecutor is instantiated as task\_doer. A parent LlmAgent called coordinator is created, also with a model and instructions. The coordinator's instructions guide it to delegate greetings to the greeter and task execution to the task\_doer. The greeter and task\_doer are added as sub-agents to the coordinator, establishing a parent-child relationship. The code then asserts that this relationship is correctly set up. Finally, it prints a message indicating that the agent hierarchy has been successfully created.

|  |
| --- |
| from google.adk.agents import LlmAgent, BaseAgent  from google.adk.agents.invocation\_context import InvocationContext  from google.adk.events import Event  from typing import AsyncGenerator  # Correctly implement a custom agent by extending BaseAgent  class TaskExecutor(BaseAgent):  """A specialized agent with custom, non-LLM behavior."""  name: str = "TaskExecutor"  description: str = "Executes a predefined task."  async def \_run\_async\_impl(self, context: InvocationContext) -> AsyncGenerator[Event, None]:  """Custom implementation logic for the task."""  # This is where your custom logic would go.  # For this example, we'll just yield a simple event.  yield Event(author=self.name, content="Task finished successfully.")  # Define individual agents with proper initialization  # LlmAgent requires a model to be specified.  greeter = LlmAgent(  name="Greeter",  model="gemini-2.0-flash-exp",  instruction="You are a friendly greeter."  )  task\_doer = TaskExecutor() # Instantiate our concrete custom agent  # Create a parent agent and assign its sub-agents  # The parent agent's description and instructions should guide its delegation logic.  coordinator = LlmAgent(  name="Coordinator",  model="gemini-2.0-flash-exp",  description="A coordinator that can greet users and execute tasks.",  instruction="When asked to greet, delegate to the Greeter. When asked to perform a task, delegate to the TaskExecutor.",  sub\_agents=[  greeter,  task\_doer  ]  )  # The ADK framework automatically establishes the parent-child relationships.  # These assertions will pass if checked after initialization.  assert greeter.parent\_agent == coordinator  assert task\_doer.parent\_agent == coordinator  print("Agent hierarchy created successfully.") |

This code excerpt illustrates the employment of the LoopAgent within the Google ADK framework to establish iterative workflows. The code defines two agents: ConditionChecker and ProcessingStep. ConditionChecker is a custom agent that checks a "status" value in the session state. If the "status" is "completed", ConditionChecker escalates an event to stop the loop. Otherwise, it yields an event to continue the loop. ProcessingStep is an LlmAgent using the "gemini-2.0-flash-exp" model. Its instruction is to perform a task and set the session "status" to "completed" if it's the final step. A LoopAgent named StatusPoller is created. StatusPoller is configured with max\_iterations=10. StatusPoller includes both ProcessingStep and an instance of ConditionChecker as sub-agents. The LoopAgent will execute the sub-agents sequentially for up to 10 iterations, stopping if ConditionChecker finds the status is "completed".

|  |
| --- |
| import asyncio  from typing import AsyncGenerator  from google.adk.agents import LoopAgent, LlmAgent, BaseAgent  from google.adk.events import Event, EventActions  from google.adk.agents.invocation\_context import InvocationContext  # Best Practice: Define custom agents as complete, self-describing classes.  class ConditionChecker(BaseAgent):  """A custom agent that checks for a 'completed' status in the session state."""  name: str = "ConditionChecker"  description: str = "Checks if a process is complete and signals the loop to stop."  async def \_run\_async\_impl(  self, context: InvocationContext  ) -> AsyncGenerator[Event, None]:  """Checks state and yields an event to either continue or stop the loop."""  status = context.session.state.get("status", "pending")  is\_done = (status == "completed")  if is\_done:  # Escalate to terminate the loop when the condition is met.  yield Event(author=self.name, actions=EventActions(escalate=True))  else:  # Yield a simple event to continue the loop.  yield Event(author=self.name, content="Condition not met, continuing loop.")  # Correction: The LlmAgent must have a model and clear instructions.  process\_step = LlmAgent(  name="ProcessingStep",  model="gemini-2.0-flash-exp",  instruction="You are a step in a longer process. Perform your task. If you are the final step, update session state by setting 'status' to 'completed'."  )  # The LoopAgent orchestrates the workflow.  poller = LoopAgent(  name="StatusPoller",  max\_iterations=10,  sub\_agents=[  process\_step,  ConditionChecker() # Instantiating the well-defined custom agent.  ]  )  # This poller will now execute 'process\_step'  # and then 'ConditionChecker'  # repeatedly until the status is 'completed' or 10 iterations  # have passed. |

This code excerpt elucidates the SequentialAgent pattern within the Google ADK, engineered for the construction of linear workflows. This code defines a sequential agent pipeline using the google.adk.agents library. The pipeline consists of two agents, step1 and step2. step1 is named "Step1\_Fetch" and its output will be stored in the session state under the key "data". step2 is named "Step2\_Process" and is instructed to analyze the information stored in session.state["data"] and provide a summary. The SequentialAgent named "MyPipeline" orchestrates the execution of these sub-agents. When the pipeline is run with an initial input, step1 will execute first. The response from step1 will be saved into the session state under the key "data". Subsequently, step2 will execute, utilizing the information that step1 placed into the state as per its instruction. This structure allows for building workflows where the output of one agent becomes the input for the next. This is a common pattern in creating multi-step AI or data processing pipelines.

|  |
| --- |
| from google.adk.agents import SequentialAgent, Agent  # This agent's output will be saved to session.state["data"]  step1 = Agent(name="Step1\_Fetch", output\_key="data")  # This agent will use the data from the previous step.  # We instruct it on how to find and use this data.  step2 = Agent(  name="Step2\_Process",  instruction="Analyze the information found in state['data'] and provide a summary."  )  pipeline = SequentialAgent(  name="MyPipeline",  sub\_agents=[step1, step2]  )  # When the pipeline is run with an initial input, Step1 will execute,  # its response will be stored in session.state["data"], and then  # Step2 will execute, using the information from the state as instructed. |

The following code example illustrates the ParallelAgent pattern within the Google ADK, which facilitates the concurrent execution of multiple agent tasks. The data\_gatherer is designed to run two sub-agents concurrently: weather\_fetcher and news\_fetcher. The weather\_fetcher agent is instructed to get the weather for a given location and store the result in session.state["weather\_data"]. Similarly, the news\_fetcher agent is instructed to retrieve the top news story for a given topic and store it in session.state["news\_data"]. Each sub-agent is configured to use the "gemini-2.0-flash-exp" model. The ParallelAgent orchestrates the execution of these sub-agents, allowing them to work in parallel. The results from both weather\_fetcher and news\_fetcher would be gathered and stored in the session state. Finally, the example shows how to access the collected weather and news data from the final\_state after the agent's execution is complete.

|  |
| --- |
| from google.adk.agents import Agent, ParallelAgent  # It's better to define the fetching logic as tools for the agents  # For simplicity in this example, we'll embed the logic in the agent's instruction.  # In a real-world scenario, you would use tools.  # Define the individual agents that will run in parallel  weather\_fetcher = Agent(  name="weather\_fetcher",  model="gemini-2.0-flash-exp",  instruction="Fetch the weather for the given location and return only the weather report.",  output\_key="weather\_data" # The result will be stored in session.state["weather\_data"]  )  news\_fetcher = Agent(  name="news\_fetcher",  model="gemini-2.0-flash-exp",  instruction="Fetch the top news story for the given topic and return only that story.",  output\_key="news\_data" # The result will be stored in session.state["news\_data"]  )  # Create the ParallelAgent to orchestrate the sub-agents  data\_gatherer = ParallelAgent(  name="data\_gatherer",  sub\_agents=[  weather\_fetcher,  news\_fetcher  ]  ) |

The provided code segment exemplifies the "Agent as a Tool" paradigm within the Google ADK, enabling an agent to utilize the capabilities of another agent in a manner analogous to function invocation. Specifically, the code defines an image generation system using Google's LlmAgent and AgentTool classes. It consists of two agents: a parent artist\_agent and a sub-agent image\_generator\_agent. The generate\_image function is a simple tool that simulates image creation, returning mock image data. The image\_generator\_agent is responsible for using this tool based on a text prompt it receives. The artist\_agent's role is to first invent a creative image prompt. It then calls the image\_generator\_agent through an AgentTool wrapper. The AgentTool acts as a bridge, allowing one agent to use another agent as a tool. When the artist\_agent calls the image\_tool, the AgentTool invokes the image\_generator\_agent with the artist's invented prompt. The image\_generator\_agent then uses the generate\_image function with that prompt. Finally, the generated image (or mock data) is returned back up through the agents. This architecture demonstrates a layered agent system where a higher-level agent orchestrates a lower-level, specialized agent to perform a task.

|  |
| --- |
| from google.adk.agents import LlmAgent  from google.adk.tools import agent\_tool  from google.genai import types  # 1. A simple function tool for the core capability.  # This follows the best practice of separating actions from reasoning.  def generate\_image(prompt: str) -> dict:  """  Generates an image based on a textual prompt.  Args:  prompt: A detailed description of the image to generate.  Returns:  A dictionary with the status and the generated image bytes.  """  print(f"TOOL: Generating image for prompt: '{prompt}'")  # In a real implementation, this would call an image generation API.  # For this example, we return mock image data.  mock\_image\_bytes = b"mock\_image\_data\_for\_a\_cat\_wearing\_a\_hat"  return {  "status": "success",  # The tool returns the raw bytes, the agent will handle the Part creation.  "image\_bytes": mock\_image\_bytes,  "mime\_type": "image/png"  }  # 2. Refactor the ImageGeneratorAgent into an LlmAgent.  # It now correctly uses the input passed to it.  image\_generator\_agent = LlmAgent(  name="ImageGen",  model="gemini-2.0-flash",  description="Generates an image based on a detailed text prompt.",  instruction=(  "You are an image generation specialist. Your task is to take the user's request "  "and use the `generate\_image` tool to create the image. "  "The user's entire request should be used as the 'prompt' argument for the tool. "  "After the tool returns the image bytes, you MUST output the image."  ),  tools=[generate\_image]  )  # 3. Wrap the corrected agent in an AgentTool.  # The description here is what the parent agent sees.  image\_tool = agent\_tool.AgentTool(  agent=image\_generator\_agent,  description="Use this tool to generate an image. The input should be a descriptive prompt of the desired image."  )  # 4. The parent agent remains unchanged. Its logic was correct.  artist\_agent = LlmAgent(  name="Artist",  model="gemini-2.0-flash",  instruction=(  "You are a creative artist. First, invent a creative and descriptive prompt for an image. "  "Then, use the `ImageGen` tool to generate the image using your prompt."  ),  tools=[image\_tool]  ) |

**At a Glance**

**What:** Complex problems often exceed the capabilities of a single, monolithic LLM-based agent. A solitary agent may lack the diverse, specialized skills or access to the specific tools needed to address all parts of a multifaceted task. This limitation creates a bottleneck, reducing the system's overall effectiveness and scalability. As a result, tackling sophisticated, multi-domain objectives becomes inefficient and can lead to incomplete or suboptimal outcomes.

**Why:** The Multi-Agent Collaboration pattern offers a standardized solution by creating a system of multiple, cooperating agents. A complex problem is broken down into smaller, more manageable sub-problems. Each sub-problem is then assigned to a specialized agent with the precise tools and capabilities required to solve it. These agents work together through defined communication protocols and interaction models like sequential handoffs, parallel workstreams, or hierarchical delegation. This agentic, distributed approach creates a synergistic effect, allowing the group to achieve outcomes that would be impossible for any single agent.

**Rule of thumb:** Use this pattern when a task is too complex for a single agent and can be decomposed into distinct sub-tasks requiring specialized skills or tools. It is ideal for problems that benefit from diverse expertise, parallel processing, or a structured workflow with multiple stages, such as complex research and analysis, software development, or creative content generation.

**Visual summary**

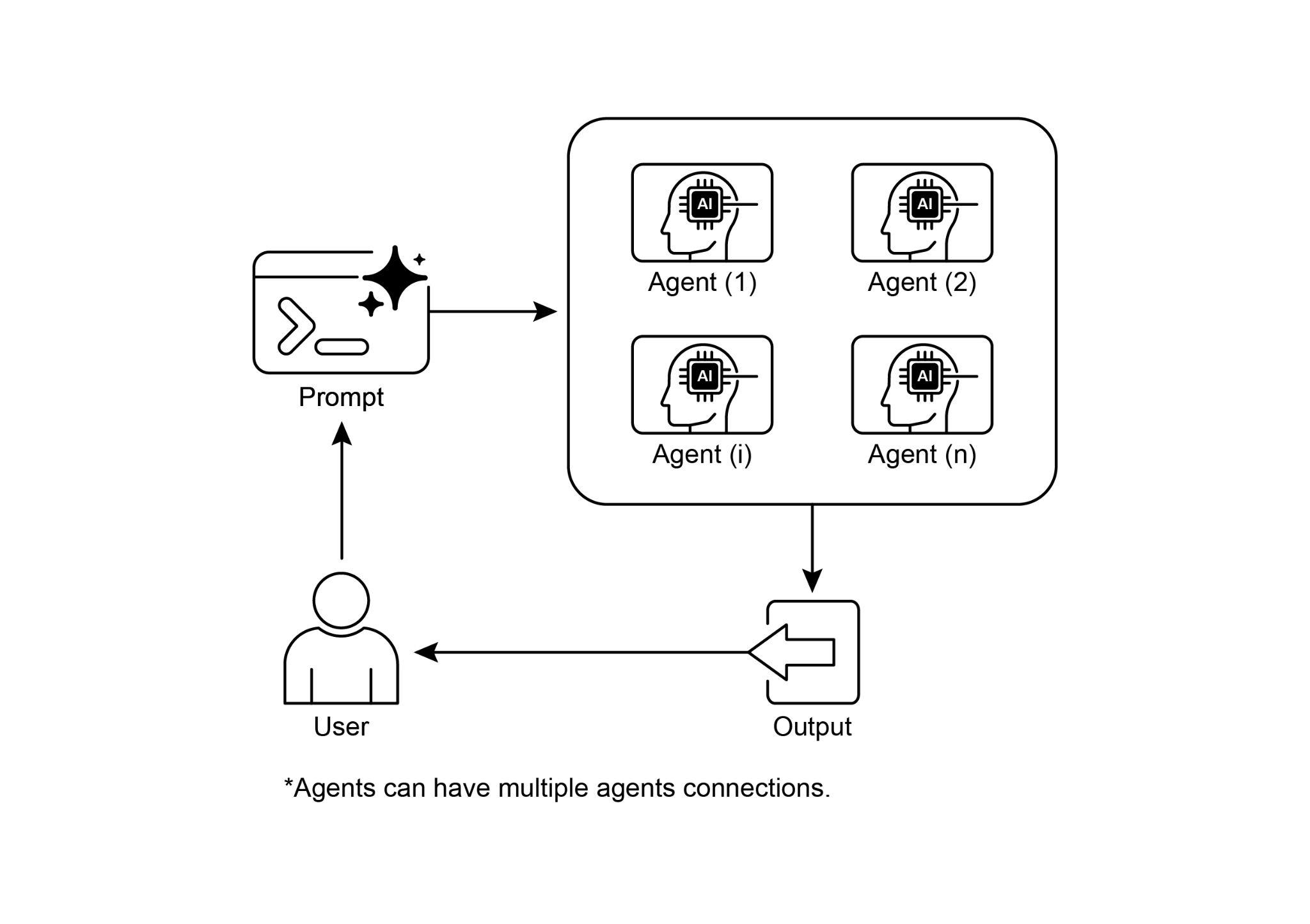


Fig.3: Multi-Agent design pattern

**Key Takeaways**

* Multi-agent collaboration involves multiple agents working together to achieve a common goal.
* This pattern leverages specialized roles, distributed tasks, and inter-agent communication.
* Collaboration can take forms like sequential handoffs, parallel processing, debate, or hierarchical structures.
* This pattern is ideal for complex problems requiring diverse expertise or multiple distinct stages.

**Conclusion**

This chapter explored the Multi-Agent Collaboration pattern, demonstrating the benefits of orchestrating multiple specialized agents within systems. We examined various collaboration models, emphasizing the pattern's essential role in addressing complex, multifaceted problems across diverse domains. Understanding agent collaboration naturally leads to an inquiry into their interactions with the external environment.

**References**

1. Multi-Agent Collaboration Mechanisms: A Survey of LLMs, <https://arxiv.org/abs/2501.06322>
2. Multi-Agent System — The Power of Collaboration, <https://aravindakumar.medium.com/introducing-multi-agent-frameworks-the-power-of-collaboration-e9db31bba1b6>

**第7章\_多智能体协作**

第7章：多智能体协作

虽然单体式智能体架构在处理定义明确的问题时可能很有效，但在面对复杂的多领域任务时，其能力往往会受到限制。多智能体协作模式通过将系统构建为不同的、专业化智能体的协作集合来解决这些局限性。这种方法基于任务分解原则，即把一个高层次的目标分解为离散的子问题。然后，每个子问题被分配给一个拥有最适合该任务的特定工具、数据访问或推理能力的智能体。

例如，一个复杂的研究查询可能会被分解，并分配给研究代理进行信息检索，分配给数据分析代理进行统计处理，分配给综合代理生成最终报告。这样一个系统的有效性不仅归因于分工，还关键取决于代理间通信的机制。这需要一个标准化的通信协议和一个共享的本体论，使代理能够交换数据、委派子任务并协调行动，以确保最终输出的连贯性。

这种分布式架构具有多种优势，包括增强的模块化、可扩展性和健壮性，因为单个智能体的故障不一定会导致整个系统的故障。这种协作能够产生协同效应，使得多智能体系统的整体性能超越集合中任何单个智能体的潜在能力。

**多智能体协作模式概述**

多智能体协作模式涉及设计这样的系统：多个独立或半独立的智能体共同协作以实现一个共同目标。每个智能体通常都有明确的角色、与总体目标一致的特定目标，并且可能可以访问不同的工具或知识库。这种模式的力量在于这些智能体之间的交互和协同作用。

协作可以采取多种形式：

* **顺序交接：**一个智能体完成一项任务，并将其输出传递给另一个智能体，以便在流水线中进行下一步操作（类似于规划模式，但明确涉及不同的智能体）。
* **并行处理：**多个代理同时处理问题的不同部分，随后将其结果合并。
* **辩论与共识：**多智能体协作，即具有不同观点和信息来源的智能体参与讨论以评估选项，最终达成共识或做出更明智的决策。
* **分层结构：**管理者代理可能会根据工具访问权限或插件功能动态地将任务委派给工作者代理，并综合他们的结果。每个代理还可以处理相关的工具组，而不是由单个代理处理所有工具。
* **专家团队：**在不同领域拥有专业知识的人员（如研究人员、作家、编辑）协作完成复杂的任务。
* **批评审查者：**代理创建初始输出，如计划、草稿或答案。然后，第二组代理会严格评估这些输出是否符合政策、安全、合规性、正确性、质量以及与组织目标的一致性。原始创建者或最终代理会根据这些反馈对输出进行修订。这种模式在代码生成、研究写作、逻辑检查和确保道德一致性方面特别有效。这种方法的优点包括增强健壮性、提高质量以及减少产生幻觉或错误的可能性。

多智能体系统（见图1）从根本上包括对智能体角色和职责的界定、建立智能体之间交换信息的通信渠道，以及制定指导其协作努力的任务流程或交互协议。

图1：多智能体系统示例

像Crew AI和谷歌ADK这样的框架，旨在通过提供用于规范智能体、任务及其交互过程的结构来促进这一范式。这种方法对于需要多种专业知识、包含多个离散阶段，或利用并发处理优势以及跨智能体信息确证的挑战特别有效。

**实际应用与用例**

多智能体协作是一种强大的模式，适用于众多领域：

* **复杂研究与分析：**一组智能体可以协作开展研究项目。一个智能体可能专门负责搜索学术数据库，另一个负责总结研究结果，第三个负责识别趋势，第四个负责将信息综合成报告。这反映了人类研究团队可能的运作方式。
* **软件开发：**设想有多个智能体协作开发软件。一个智能体可以是需求分析师，另一个是代码生成器，第三个是测试员，第四个是留档编写员。他们可以相互传递输出结果，以构建和验证组件。
* **创意内容生成：**创建营销活动可能涉及市场调研专员、广告文字撰写人、平面设计专员（使用图像生成工具）和社交媒体日程安排专员，他们共同协作。
* **金融分析：**多智能体系统可以分析金融市场。智能体可能专门负责获取股票数据、分析新闻情绪、进行技术分析以及生成投资建议。
* **客户支持升级：**一线支持人员可以处理初始咨询，必要时将复杂问题升级给专业代理（例如技术专家或计费专家），展示了基于问题复杂性的顺序交接。
* **供应链优化：**智能体可以代表供应链中的不同节点（供应商、制造商、经销商），并协作优化库存水平、物流和调度，以应对不断变化的需求或中断。
* **网络分析与修复**：自主操作从智能体架构中受益匪浅，尤其是在故障定位方面。多个智能体可以协作对问题进行分类和修复，并提出最优行动方案。这些智能体还可以与传统机器学习模型和工具集成，在利用现有系统的同时，提供生成式AI的优势。

能够界定专门的智能体并精心编排它们之间的相互关系，使开发者能够构建出具有更高模块化、可扩展性的系统，并且能够应对单一集成智能体无法解决的复杂问题。

**多智能体协作：探索相互关系与通信结构**

理解智能体之间复杂的交互和通信方式，是设计有效多智能体系统的基础。如图2所示，存在一系列相互关系和通信模型，从最简单的单智能体场景到复杂的、定制设计的协作框架。每个模型都有独特的优势和挑战，影响着多智能体系统的整体效率、鲁棒性和适应性。

**1. 单智能体：**在最基本的层面上，“单智能体”自主运行，无需与其他实体进行直接交互或通信。虽然这种模型易于实现和管理，但其能力本质上受限于单个智能体的范围和资源。它适用于可分解为独立子问题的任务，每个子问题都可由单个自给自足的智能体解决。

**2. 网络模式：**“网络”模式是迈向协作的重要一步，在这种模式下，多个主体以去中心化的方式直接相互交互。通信通常以点对点的方式进行，从而实现信息、资源甚至任务的共享。这种模式增强了系统的弹性，因为一个主体的故障不一定会使整个系统瘫痪。然而，在一个庞大且无结构的网络中管理通信开销并确保连贯的决策制定可能具有挑战性。

**3. 监督者：**在“监督者”模型中，一个专门的代理，即“监督者”，负责监督和协调一组下属代理的活动。监督者充当通信、任务分配和冲突解决的中心枢纽。这种层级结构提供了明确的权力线，能够简化管理和控制。然而，它引入了单点故障（监督者），如果监督者被大量下属或复杂任务压垮，可能会成为瓶颈。

**4. 作为工具的监督者：**此模型是“监督者”概念的细致延伸，在该模型中，监督者的角色较少涉及直接的命令和控制，而更多地是为其他主体提供资源、指导或分析支持。监督者可能会提供工具、数据或计算服务，使其他主体能够更有效地执行任务，而不一定对其每一个行动都发号施令。这种方法旨在利用监督者的能力，同时避免实施僵化的自上而下的控制。

**5. 分层式：**“分层式”模型在监督者概念的基础上进行扩展，以创建一个多层级的组织结构。这涉及多个层级的监督者，高层监督者监督低层监督者，最终在最底层是一组操作代理。这种结构非常适合可分解为子问题的复杂问题，每个子问题由层级结构中的特定层进行管理。它为可扩展性和复杂性管理提供了一种结构化方法，允许在既定边界内进行分布式决策。

图2：智能体以各种方式进行通信和交互。

**6. 自定义：**“自定义”模型代表了多智能体系统设计中的终极灵活性。它允许创建独特的相互关系和通信结构，这些结构是根据给定问题或应用的特定需求精确定制的。这可能涉及结合前面提到的模型元素的混合方法，或者是源于环境的unique约束和机会的全新设计。自定义模型通常源于优化特定性能指标、处理高度动态环境或将特定领域知识融入系统架构的需求。设计和实现自定义模型通常需要深入理解多智能体系统原理，并仔细考虑通信协议、协调机制和涌现行为。

总之，多智能体系统的交互关系和通信模型的选择是一项关键的设计决策。每种模型都有其独特的优缺点，而最优选择取决于任务的复杂性、智能体的数量、期望的自主程度、对鲁棒性的需求以及可接受的通信开销等因素。多智能体系统未来的发展可能会继续探索和完善这些模型，同时也会为协作智能开发新的范式。

**实践代码（Crew AI）**

这段Python代码使用CrewAI框架定义了一个由AI驱动的团队，用于生成一篇关于AI趋势的博客文章。它首先设置环境，从.env文件中加载API密钥。应用程序的核心是定义两个代理：一个研究人员负责查找和总结AI趋势，另一个作者根据研究结果撰写博客文章。

相应地定义了两项任务：一项是研究趋势，另一项是撰写博客文章，其中撰写任务依赖于研究任务的输出。然后将这些智能体和任务组合成一个团队，指定一个按顺序执行任务的流程。团队使用智能体、任务和语言模型（具体为“gemini-2.0-flash”模型）进行初始化。主函数使用 kickoff() 方法执行这个团队，协调智能体之间的协作以产生预期的输出。最后，代码打印出团队执行的最终结果，即生成的博客文章。

|  |
| --- |
| 导入 os  from dotenv import load\_dotenv  从 crewai 导入 Agent、Task、Crew、Process  从 langchain\_google\_genai 导入 ChatGoogleGenerativeAI  def setup\_environment():  加载环境变量并检查是否存在所需的 API 密钥。  load\_dotenv()  if not os.getenv("GOOGLE\_API\_KEY"):  raise ValueError("未找到 GOOGLE\_API\_KEY，请在.env 文件中设置。")  def main():  """  使用最新的Gemini模型初始化并运行用于内容生产的AI团队。  """  setup\_environment()  # 定义要使用的语言模型。  # 更新为Gemini 2.0系列的模型，以获得更好的性能和特性。  # 对于前沿（预览）功能，您可以使用 "gemini-2.5-flash"。  llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")  # 定义具有特定角色和目标的智能体  researcher = Agent(  role='高级研究分析师',  goal='查找并总结AI领域的最新趋势。',  背景故事="你是一位经验丰富的研究分析师，擅长识别关键趋势并综合信息。",  verbose=True,  allow\_delegation=False,  )  writer = Agent(  role='技术内容撰写员',  目标='根据研究结果撰写一篇清晰且引人入胜的博客文章'。  背景故事="你是一位技艺娴熟的作家，能够将复杂的技术主题转化为通俗易懂的内容。",  verbose=True,  allow\_delegation=False,  )  # 为代理定义任务  research\_task = Task(  描述="研究2024 - 2025年人工智能领域的前三大新兴趋势。重点关注实际应用和潜在影响。",  预期输出="前三大AI趋势的详细总结，包括要点和来源。",  代理=研究员，  )  writing\_task = Task(  描述="根据研究结果撰写一篇500字的博客文章。文章应引人入胜，易于普通读者理解。",  预期输出为一篇完整的500字博客文章，内容关于最新的AI趋势。  代理=作者,  context=[research\_task],  )  # 创建团队  blog\_creation\_crew = Crew(  agents=[研究员, 作者],  tasks=[research\_task, writing\_task],  process=Process.sequential,  llm=llm,  verbose=2 # 设置详细程度以获取详细的机组执行日志  )  # 处决船员  print("## 使用Gemini 2.0 Flash运行博客创建团队... ##")  try:  result = blog\_creation\_crew.kickoff()  print("\n------------------\n")  print("## 机组最终输出 ##")  print(result)  except Exception as e:  print(f"\n发生了意外错误: {e}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

我们现在将深入探讨Google ADK框架内的更多示例，特别着重于分层、并行和顺序协调范式，以及将代理作为操作工具的实现。

**实践代码（谷歌ADK）**

以下代码示例展示了如何通过创建父子关系在Google ADK中建立分层代理结构。代码定义了两种类型的代理：LlmAgent和从BaseAgent派生的自定义TaskExecutor代理。TaskExecutor专为特定的非大语言模型（LLM）任务而设计，在本示例中，它仅产生一个“任务成功完成”事件。一个名为greeter的LlmAgent使用指定的模型和指令进行初始化，以充当友好的问候者。自定义的TaskExecutor被实例化为task\_doer。一个名为coordinator的父LlmAgent也使用模型和指令创建。coordinator的指令指导它将问候任务委派给greeter，并将任务执行委派给task\_doer。greeter和task\_doer作为子代理添加到coordinator中，从而建立了父子关系。代码随后断言这种关系已正确设置。最后，它打印一条消息，表明代理层次结构已成功创建。

|  |
| --- |
| from google.adk.agents import LlmAgent, BaseAgent  from google.adk.agents.invocation\_context import InvocationContext  从google.adk.events导入Event  从 typing 导入 AsyncGenerator  # 通过扩展 BaseAgent 正确实现自定义代理  class TaskExecutor(BaseAgent):  """具有自定义、非大语言模型行为的专业代理。"""  name: str = "任务执行器"  description: str = "执行预定义任务。"  async def \_run\_async\_impl(self, context: InvocationContext) -> AsyncGenerator[Event, None]:  """任务的自定义实现逻辑。"""  # 这里是你自定义逻辑的位置。  # 对于这个示例，我们将只生成一个简单的事件。  yield Event(author=self.name, content="任务已成功完成。")  # 定义经过适当初始化的个体智能体  # LlmAgent需要指定一个模型。  greeter = LlmAgent(  name="Greeter",  model="gemini-2.0-flash-exp",  instruction="你是一位友好的迎宾员。"  )  task\_doer = TaskExecutor() # 实例化我们具体的自定义代理  # 创建一个父代理并分配其子代理  # 父代理的描述和指令应指导其委派逻辑。  coordinator = LlmAgent(  name="协调员",  model="gemini-2.0-flash-exp",  description="一个可以向用户打招呼并执行任务的协调器。",  instruction="当被要求打招呼时，委托给问候器。当被要求执行任务时，委托给任务执行器。",  sub\_agents=[  迎宾员  任务执行者  ]  )  # ADK框架会自动建立父子关系。  # 这些断言在初始化后检查时将通过。  断言问候者的父代理等于协调器  断言任务执行者的父代理等于协调器  print("代理层级创建成功。") |

这段代码片段展示了如何在Google ADK框架中使用LoopAgent来建立迭代工作流。代码定义了两个代理：ConditionChecker和ProcessingStep。ConditionChecker是一个自定义代理，用于检查会话状态中的“status”值。如果“status”为“completed”，ConditionChecker会触发一个事件来停止循环；否则，它会触发一个事件来继续循环。ProcessingStep是一个使用“gemini-2.0-flash-exp”模型的LlmAgent。它的指令是执行一项任务，并在最后一步将会话“status”设置为“completed”。创建了一个名为StatusPoller的LoopAgent。StatusPoller被配置为最大迭代次数为10次。StatusPoller包含ProcessingStep和ConditionChecker的一个实例作为子代理。LoopAgent将按顺序执行子代理，最多迭代10次，如果ConditionChecker发现状态为“completed”，则停止执行。

|  |
| --- |
| import asyncio  从 typing 导入 AsyncGenerator  from google.adk.agents import LoopAgent, LlmAgent, BaseAgent  从google.adk.events导入Event, EventActions  from google.adk.agents.invocation\_context import InvocationContext  #最佳实践：将自定义代理定义为完整、自描述的类。  class ConditionChecker(BaseAgent):  一个自定义代理，用于检查会话状态中是否存在“已完成”状态。  name: str = "ConditionChecker"  description: str = "检查进程是否完成，并向循环发送停止信号。"  async def \_run\_async\_impl(  self, context: InvocationContext  ) -> AsyncGenerator[Event, None]:  """检查状态并产生一个事件，以继续或停止循环。"""  status = context.session.state.get("status", "pending")  is\_done = (status == "completed")  if is\_done:  # 当条件满足时升级以终止循环。  yield Event(author=self.name, actions=EventActions(escalate=True))  否则:  # 产生一个简单事件以继续循环。  yield Event(作者=self.name, 内容="条件未满足，继续循环。")  #更正：LlmAgent必须有一个模型和明确的指令。  process\_step = LlmAgent(  name="处理步骤",  model="gemini-2.0-flash-exp",  instruction="你是一个较长流程中的一个步骤。执行你的任务。如果你是最后一步，请通过将 'status' 设置为 'completed' 来更新会话状态。"  )  # LoopAgent编排工作流程。  poller = LoopAgent(  name="状态轮询器",  max\_iterations=10,  sub\_agents=[  处理步骤  ConditionChecker() # 实例化定义良好的自定义代理。  ]  )  #此轮询器现在将执行'process\_step'  # 然后是 'ConditionChecker'  # 重复执行，直到状态为 '已完成' 或达到 10 次迭代  #已经过去了。 |

这段代码片段阐释了谷歌ADK中的顺序代理模式，该模式专为构建线流而设计。此代码使用google.adk.agents库定义了一个顺序代理管道。该管道由两个代理组成，即step1和step2。step1名为“Step1\_Fetch”，其输出将以键“data”存储在会话状态中。step2名为“Step2\_Process”，其任务是分析会话状态中键“data”下存储的信息并提供摘要。名为“MyPipeline”的顺序代理负责协调这些子代理的执行。当管道以初始输入运行时，step1将首先执行。step1的响应将以键“data”保存到会话状态中。随后，step2将执行，按照其指令利用step1存储在状态中的信息。这种结构允许构建工作流，其中一个代理的输出成为下一个代理的输入。这是创建多步骤AI或数据处理管道时的常见模式。

|  |
| --- |
| from google.adk.agents import SequentialAgent, Agent  # 此代理的输出将保存到 session.state["data"]  step1 = Agent(name="Step1\_Fetch", output\_key="data")  #此代理将使用上一步的数据。  #我们指导它如何查找和使用这些数据。  step2 = Agent(  name="Step2\_Process",  instruction="分析在state['data']中找到的信息并提供摘要。"  )  pipeline = SequentialAgent(  name="我的管道",  sub\_agents=[step1, step2]  )  #当管道以初始输入运行时，步骤1将执行  # 其响应将存储在 session.state["data"] 中，然后  # 步骤2将执行，按照指示使用状态中的信息。 |

以下代码示例展示了Google ADK中的ParallelAgent模式，该模式有助于并发执行多个代理任务。data\_gatherer被设计为同时运行两个子代理：weather\_fetcher和news\_fetcher。weather\_fetcher代理被指示获取给定地点的天气信息，并将结果存储在session.state["weather\_data"]中。同样，news\_fetcher代理被指示检索给定主题的头条新闻，并将其存储在session.state["news\_data"]中。每个子代理都配置为使用“gemini-2.0-flash-exp”模型。ParallelAgent负责协调这些子代理的执行，使它们能够并行工作。weather\_fetcher和news\_fetcher的结果将被收集并存储在会话状态中。最后，示例展示了在代理执行完成后，如何从final\_state中访问收集到的天气和新闻数据。

|  |
| --- |
| 从google.adk.agents导入Agent, ParallelAgent  #最好将获取逻辑定义为代理的工具  # 为了简化这个示例，我们将把逻辑嵌入到代理的指令中。  # 在现实场景中，你会使用工具。  # 定义将并行运行的各个代理  weather\_fetcher = Agent(  name="weather\_fetcher",  model="gemini-2.0-flash-exp",  instruction="获取给定地点的天气信息，仅返回天气报告。",  output\_key="weather\_data" # 结果将存储在 session.state["weather\_data"] 中  )  news\_fetcher = Agent(  name="news\_fetcher",  model="gemini-2.0-flash-exp",  instruction="获取给定主题的头条新闻报道，并仅返回该报道。",  output\_key="news\_data" # 结果将存储在 session.state["news\_data"] 中  )  # 创建并行代理以协调子代理  data\_gatherer = ParallelAgent(  name="数据收集器",  sub\_agents=[  weather\_fetcher,  新闻抓取器  ]  ) |

所提供的代码片段展示了 Google ADK 中 “代理作为工具” 的范式，使一个代理能够以类似于函数调用的方式利用另一个代理的功能。具体来说，该代码使用 Google 的 LlmAgent 和 AgentTool 类定义了一个图像生成系统。它由两个代理组成：父代理 artist\_agent 和子代理 image\_generator\_agent。generate\_image 函数是一个简单的工具，用于模拟图像创建，返回模拟图像数据。image\_generator\_agent 负责根据接收到的文本提示使用这个工具。artist\_agent 的作用是首先构思一个富有创意的图像提示，然后通过 AgentTool 包装器调用 image\_generator\_agent。AgentTool 充当桥梁，允许一个代理将另一个代理作为工具使用。当 artist\_agent 调用 image\_tool 时，AgentTool 会使用 artist 构思的提示来调用 image\_generator\_agent。image\_generator\_agent 随后使用该提示调用 generate\_image 函数。最后，生成的图像（或模拟数据）通过代理返回。这种架构展示了一个分层的代理系统，其中高层代理协调低层的专业代理来执行任务。

|  |
| --- |
| from google.adk.agents import LlmAgent  from google.adk.tools import agent\_tool  从google.genai导入types  # 1. 核心能力的简单功能工具。  #这遵循了将行动与推理分开的最佳实践。  def generate\_image(prompt: str) -> dict:  """  根据文本提示生成图像。  参数：  提示：对要生成的图像的详细描述。  返回值：  一个包含状态和生成图像字节的字典。  """  print(f"工具：正在为提示词 '{prompt}' 生成图像")  # 在实际实现中，这将调用图像生成 API。  # 对于此示例，我们返回模拟图像数据。  mock\_image\_bytes = b"mock\_image\_data\_for\_a\_cat\_wearing\_a\_hat"  返回 {  "状态": "成功",  该工具返回原始字节，代理将处理部件创建。  "image\_bytes": mock\_image\_bytes,  "mime\_type": "image/png"  }  # 2. 将 ImageGeneratorAgent 重构为 LlmAgent。  #现在它能正确使用传递给它的输入。  image\_generator\_agent = LlmAgent(  name="图像生成",  model="gemini-2.0-flash",  description="根据详细的文本提示生成图像。",  instruction=(  "你是一位图像生成专家。你的任务是接收用户的请求 "  并使用`generate\_image`工具创建图像。  用户的整个请求应作为工具的'prompt'参数使用。  工具返回图像字节后，你必须输出图像。  ),  工具=[生成图像]  )  # 3. 将修正后的智能体封装在 AgentTool 中。  #这里的描述是父代理看到的内容。  image\_tool = agent\_tool.AgentTool(  agent=图像生成器代理,  description="使用此工具生成图像。输入应为所需图像的描述性提示。"  )  # 4. 父代理保持不变。其逻辑是正确的。  artist\_agent = LlmAgent(  name="艺术家",  model="gemini-2.0-flash",  instruction=(  "你是一位富有创造力的艺术家。首先，为一幅图像构思一个富有创意且生动的提示词。"  然后，使用 `ImageGen` 工具根据你的提示生成图像。  ),  tools=[image\_tool]  ) |

**概览**

**问题描述：**复杂问题往往超出了单一、整体式基于大语言模型（LLM）的智能体的能力范围。单个智能体可能缺乏解决多方面任务所需的多样化、专业化技能，或无法获取特定工具。这一局限性形成了瓶颈，降低了系统的整体有效性和可扩展性。因此，处理复杂的多领域目标变得低效，可能导致结果不完整或不理想。

**原因：**多智能体协作模式通过创建一个由多个协作智能体组成的系统，提供了一种标准化的解决方案。一个复杂的问题被分解为更小、更易于管理的子问题。然后，每个子问题被分配给一个具备解决该问题所需精确工具和能力的专业智能体。这些智能体通过定义好的通信协议和交互模型（如顺序交接、并行工作流或分层委派）协同工作。这种基于智能体的分布式方法产生了协同效应，使整个团队能够实现任何单个智能体都无法实现的成果。

**经验法则：**当一项任务对于单个智能体来说过于复杂，且可以分解为需要专业技能或工具的不同子任务时，使用此模式。它非常适合那些受益于多样化专业知识、并行处理或具有多阶段结构化工作流程的问题，如复杂的研究与分析、软件开发或创意内容生成。

**可视化总结**

图3：多智能体设计模式

**要点总结**

* 多智能体协作涉及多个智能体共同努力以实现一个共同目标。
* 这种模式利用专门的角色、分布式任务和智能体间通信。
* 协作可以采取顺序交接、并行处理、辩论或层级结构等形式。
* 这种模式非常适合需要不同专业知识或多个不同阶段的复杂问题。

**结论**

本章探讨了多智能体协作模式，展示了在系统中协调多个专业智能体的益处。我们研究了各种协作模型，强调了该模式在解决跨不同领域的复杂、多方面问题中的关键作用。理解智能体协作自然会引发对其与外部环境交互的探究。

**参考文献**

1. 多智能体协作机制：大语言模型综述<https://arxiv.org/abs/2501.06322>
2. 多智能体系统 — 协作的力量<https://aravindakumar.medium.com/introducing-multi-agent-frameworks-the-power-of-collaboration-e9db31bba1b6>