

AutoGen生态系统提供了你需要创建AI代理所需要的一切，特别是多agent工作流，框架，开发工具和应用程序。

框架层进行了分层设计和可扩展性设计。每一层有清晰的任务，并且为上层服务，你可以使用不同层的API进行设计和开发。

1. core api 实现了消息传递，事件驱动agent和本地的运行时动态分发，它同时支持python和.net.
2. agent chat api,实现了一个简单的内部使用的API用于快速开发原型，这个API在coreapi之上，支持类似多用户聊天的场景。
3. extensions api ，允许三方集成和扩展framework能力，支持指定的LLM客户端集成如OPENAI AzureOpenAI支持代码执行能力。

生态系统支持两个必要的开发工具：

Autogen studio：提供零代码的GUI编程方式搭建多智能体应用程序。

Autogen bench：提供用于评估agent性能的标准化管理套件。

我们可以使用AutoGen框架和开发工具来创建自己领域的应用程序，例如magentic-one是一个最先进的多智能体团队构建器，它使用AgentChat API和扩展API，可以处理各式各样的任务，例如浏览网页，代码执行，文件处理。

<https://blog.csdn.net/qq_41472205/category_12838949.html>

Autogen 中 agent 各个组成部分的设计思路大致为：

### Abilities & Prior Knowledge

* **逻辑思考能力：**

和大多数[智能体设计](https://zhida.zhihu.com/search?content_id=243646070&content_type=Article&match_order=1&q=%E6%99%BA%E8%83%BD%E4%BD%93%E8%AE%BE%E8%AE%A1&zhida_source=entity)一样，Autogen 使用了语言模型（LLM）作为逻辑单元。它通过调用 OpenAI 的 Python SDK 来访问 LLM 服务。因此，任何部署了 OpenAI Compatible API 的 LLM 都可以无缝对接到 Autogen。

* **工具选择的能力**

Autogen 利用了 [openai 提供的 Tool（Functions）功能](https://link.zhihu.com/?target=https%3A//platform.openai.com/docs/api-reference/chat/create) 来调用函数，而不是使用自定义提示来引导逻辑模型选择工具。调用 Openai 服务时，在请求体的 tools 中提供候选函数的信息，Openai API 就会从中选出最可能实现用户诉求的函数。

* **通信功能**

每个 agent 都可以使用自带的 [send](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/5a5c0f2a11bfc1efe3955fa9f7e2ec38aedef14e/autogen/agentchat/conversable_agent.py%23L611) 和 [receive](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/5a5c0f2a11bfc1efe3955fa9f7e2ec38aedef14e/autogen/agentchat/conversable_agent.py%23L785C9-L785C16) 方法和其他 agent 进行通信。

### Action & Stimuli

智能体在选择了合适的工具后，可以执行该工具并接收工具返回的结果。

### Goals/Preference

Autogen 通过为不同的智能体提供不同的系统提示词来实现个性化设置。

### Past Experience

每个智能体都会维护自己的历史记录，以 List[Message] 的形式保存，其中 Message 包括了历史对话信息和执行函数的结果等

ConversableAgent：通用会话聊天代理，system\_message 会话的提示词，is\_termination\_msg判断会话是否结束，code\_execution\_config执行代码的配置，llm\_config LLM配置，其中human\_input\_mode有三种模式，ALWAYS：每次收到消息时提示用户，此时会话暂停 TERMINATE：当自动回复达到了max\_consecutive\_auto\_reply的次数才提示用户，会话暂停，NEVER：从不提示用户。通过调用generate\_reply方法启动

**UserProxyAgent 类的定义**：这是一个模拟人类用户交互的代理类，接收用户输入并根据 input\_func（输入函数）来处理输入。

通过调用以下方法启动chat\_result = user\_proxy.initiate\_chat(assistant, message="What is (44232 + 13312)) \* 5?")

AssistantAgent继承自ConversableAgent，被设计用来调用LLM处理任务，可以在执行的过程中智能调用工具，运行代码，检测代码，修复代码。register\_for\_llm方法可以用来关联函数，后续做function calling

代码执行器的思路为

1. 首先在 self.\_reply\_func\_list 插入 [\_generate\_code\_execution\_reply\_using\_executor](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/5a5c0f2a11bfc1efe3955fa9f7e2ec38aedef14e/autogen/agentchat/conversable_agent.py%23L1409) 函数。
2. 以上函数会先使用 [markdown 解析器](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/5a5c0f2a11bfc1efe3955fa9f7e2ec38aedef14e/autogen/coding/local_commandline_code_executor.py%23L180)来从输入的 message 种提取代码。
3. 而后使用[执行器](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/5a5c0f2a11bfc1efe3955fa9f7e2ec38aedef14e/autogen/coding/local_commandline_code_executor.py%23L236)来执行抽取出来的代码。

**Multi-Agent**

通过以上 initiate\_chat 我们大致清楚了 2 个 Agent 是怎么对话的：

* 通过 send 和 receive 函数实现闭环对话。

那如果我们想要让 2 个以上的 agent 进行对话，可以使用 Autogen 包装好的 [GroupChat](https://link.zhihu.com/?target=https%3A//github.com/microsoft/autogen/blob/main/notebook/agentchat_groupchat.ipynb)。

GroupChat 的实现逻辑大致为：

* 建立一个 GroupManager 来扮演邮差的角色。
* 所有 Group 里的 Agent 不直接对话，而是通过 GroupManager 实现对话的传递。

这种GroupChat 的设计方式允许我们设计自由讨论的 Multi-agent，同时也可以实现固定 flow 的 Multi-Agent 工作流。

https://zhuanlan.zhihu.com/p/699819907

# Messages

Autogen支持不同类型的message，用于智能体和节点之间通信，这些消息分两大类：agent与agent之间的通信和内部事件。

message常见的有TextMessage和MultiModalMessage：

text\_message = TextMessage(content="Hello, world!", source="User")

multi\_modal\_message = MultiModalMessage(content=["Can you describe the content of this image?", img], source="User")

### Internal Events

[ToolCallRequestEvent](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.messages.html#autogen_agentchat.messages.ToolCallRequestEvent),对tool发起请求，[ToolCallExecutionEvent](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.messages.html#autogen_agentchat.messages.ToolCallExecutionEvent),tool返回的事件。

# Define a tool that searches the web for information.

async def web\_search(query: str) -> str:

"""Find information on the web"""

return "AutoGen is a programming framework for building multi-agent applications."

# Create an agent that uses the OpenAI GPT-4o model.

model\_client = OpenAIChatCompletionClient(

model="gpt-4o",

# api\_key="YOUR\_API\_KEY",

)

agent = AssistantAgent(

name="assistant",

model\_client=model\_client,

tools=[web\_search],

system\_message="Use tools to solve tasks.",

)

# Agents

agent由以下成员和方法组成，

Name, [description](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.description)

[on\_messages()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_messages)[on\_messages\_stream()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_messages_stream):发送一串[ChatMessage](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.messages.html#autogen_agentchat.messages.ChatMessage) 得到一个[Response](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.base.html#autogen_agentchat.base.Response)，这个方法发的是最新消息，而不是历史消息。

[on\_reset()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_reset):恢复agent到初始状态

[run()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.run) and [run\_stream()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.run_stream):统一接口方法，内部调用[on\_messages()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_messages)[on\_messages\_stream()](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_messages_stream):

async def assistant\_run() -> None:

response = await agent.on\_messages(

[TextMessage(content="Find information on AutoGen", source="user")],

cancellation\_token=CancellationToken(),

)

print(response.inner\_messages)

print(response.chat\_message)

# Use asyncio.run(assistant\_run()) when running in a script.

await assistant\_run()

async def assistant\_run\_stream() -> None:

# Option 1: read each message from the stream (as shown in the previous example).

# async for message in agent.on\_messages\_stream(

# [TextMessage(content="Find information on AutoGen", source="user")],

# cancellation\_token=CancellationToken(),

# ):

# print(message)

# Option 2: use Console to print all messages as they appear.

await Console(

agent.on\_messages\_stream(

[TextMessage(content="Find information on AutoGen", source="user")],

cancellation\_token=CancellationToken(),

)

)

# Use asyncio.run(assistant\_run\_stream()) when running in a script.

await assistant\_run\_stream()

## Using Tools

AssistantAgent可以使用tools来完成一些特定的需求，web\_search tool 就是其中一个，如果需要自定义tool,必须继承BaseTool类。默认情况下，AssistantAgent执行一个tool,tool的输出会在response中以[ToolCallSummaryMessage](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.messages.html#autogen_agentchat.messages.ToolCallSummaryMessage)的形式返回。如果你的tool没有返回一个格式很好的string，你需要通过在AssistantAgent的构造方法里面设置reflect\_on\_tool\_use=True

来添加一个reflection 步骤来总结tool的输出。

### Langchain Tools

我们也可以通过[LangChainToolAdapter](https://microsoft.github.io/autogen/dev/reference/python/autogen_ext.tools.langchain.html#autogen_ext.tools.langchain.LangChainToolAdapter)来封装langchain tool使用

df = pd.read\_csv("https://raw.githubusercontent.com/pandas-dev/pandas/main/doc/data/titanic.csv")

tool = LangChainToolAdapter(PythonAstREPLTool(locals={"df": df}))

model\_client = OpenAIChatCompletionClient(model="gpt-4o")

agent = AssistantAgent(

"assistant", tools=[tool], model\_client=model\_client, system\_message="Use the `df` variable to access the dataset."

)

await Console(

agent.on\_messages\_stream(

[TextMessage(content="What's the average age of the passengers?", source="user")], CancellationToken()

)

)

## Using Model Context

AssistantAgent有一个model\_context参数，能够被用来传递给[ChatCompletionContext](https://microsoft.github.io/autogen/dev/reference/python/autogen_core.model_context.html#autogen_core.model_context.ChatCompletionContext)对象。这允许agent使用不同模型的上下文，例如[BufferedChatCompletionContext](https://microsoft.github.io/autogen/dev/reference/python/autogen_core.model_context.html#autogen_core.model_context.BufferedChatCompletionContext)来限制发送给模型的上下文。默认情况下，AssistantAgent使用[UnboundedChatCompletionContext](https://microsoft.github.io/autogen/dev/reference/python/autogen_core.model_context.html#autogen_core.model_context.UnboundedChatCompletionContext)发送完整的对话历史到model。为了限制上下文只取最近的几条聊天记录，使用[BufferedChatCompletionContext](https://microsoft.github.io/autogen/dev/reference/python/autogen_core.model_context.html#autogen_core.model_context.BufferedChatCompletionContext)。

agent = AssistantAgent(

name="assistant",

model\_client=model\_client,

tools=[web\_search],

system\_message="Use tools to solve tasks.",

model\_context=BufferedChatCompletionContext(buffer\_size=5),

# Only use the last 5 messages in the context.

)

其他预制的agent：

[UserProxyAgent](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.UserProxyAgent): 接收用户输入，并返回response

[CodeExecutorAgent](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.CodeExecutorAgent): 可以运行用户代码的agent

[OpenAIAssistantAgent](https://microsoft.github.io/autogen/dev/reference/python/autogen_ext.agents.openai.html#autogen_ext.agents.openai.OpenAIAssistantAgent): 支持 OpenAI Assistant的agent ,可以调用自定义tools.

[MultimodalWebSurfer](https://microsoft.github.io/autogen/dev/reference/python/autogen_ext.agents.web_surfer.html#autogen_ext.agents.web_surfer.MultimodalWebSurfer): 一个可以搜索网页和查看里面信息的多模态智能体。

[FileSurfer](https://microsoft.github.io/autogen/dev/reference/python/autogen_ext.agents.file_surfer.html#autogen_ext.agents.file_surfer.FileSurfer): 一个可以搜索和浏览本地文件信息的智能体。

[VideoSurfer](https://microsoft.github.io/autogen/dev/reference/python/autogen_ext.agents.video_surfer.html#autogen_ext.agents.video_surfer.VideoSurfer):一个可以观看video信息的智能体。

# Teams

[RoundRobinGroupChat](https://microsoft.github.io/autogen/dev/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.RoundRobinGroupChat)是一个有效的team配置，在这里面所有的agent分享共同的上下文，并且在循环队列中轮流地响应。每个agent在自己的回合里广播自己的response给其他agent,保证整个team维护一个一致的上下文。

# Create the primary agent.

primary\_agent = AssistantAgent(

"primary",

model\_client=model\_client,

system\_message="You are a helpful AI assistant.",

)

# Create the critic agent.

critic\_agent = AssistantAgent(

"critic",

model\_client=model\_client,

system\_message="Provide constructive feedback. Respond with 'APPROVE' to when your feedbacks are addressed.",

)

# Define a termination condition that stops the task if the critic approves.

text\_termination = TextMentionTermination("APPROVE")

# Create a team with the primary and critic agents.

team = RoundRobinGroupChat([primary\_agent, critic\_agent], termination\_condition=text\_termination)

使用run方法用task启动team

# Use `asyncio.run(...)` when running in a script.

result = await team.run(task="Write a short poem about the fall season.")

## Observing a Team

通过调用[run\_stream()](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.BaseGroupChat.run_stream)方法可以返回一系列包含message的yield生成器，最终的message是TaskResult类型：

await team.reset() # Reset the team for a new task.

async for message in team.run\_stream(task="Write a short poem about the fall season."): # type: ignore

if isinstance(message, TaskResult):

print("Stop Reason:", message.stop\_reason)

else:

print(message)

如上，可以通过检查stop\_reason来查看team停止的原因。

## Stopping a Team

通过设置[TextMentionTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.TextMentionTermination) 可以在内部停止team task。你还可以通过设置ExternalTermination在外部停止team task。

external\_termination = ExternalTermination()

team = RoundRobinGroupChat(

[primary\_agent, critic\_agent],

termination\_condition=external\_termination | text\_termination, # Use the bitwise OR operator to combine conditions.

)

# Run the team in a background task.

run = asyncio.create\_task(Console(team.run\_stream(task="Write a short poem about the fall season.")))

# Wait for some time.

await asyncio.sleep(0.1)

# Stop the team.

external\_termination.set()

# Wait for the team to finish.

await run

[ExternalTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.ExternalTermination)调用时，会等待当前agent执行完毕后再广播停止消息给所有team agent最后结束task

## Resuming a Team

可以通过调用[run()](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.BaseGroupChat.run)或者[run\_stream()](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.BaseGroupChat.run_stream)来继续执行team任务，这样不会新启动一个task

## Aborting a Team

通过[CancellationToken](https://microsoft.github.io/autogen/stable/reference/python/autogen_core.html#autogen_core.CancellationToken)来取消一个正在运行中的team task。

cancellation\_token = CancellationToken()

# Use another coroutine to run the team.

run = asyncio.create\_task(

team.run(

task="Translate the poem to Spanish.",

cancellation\_token=cancellation\_token,

)

)

# Cancel the run.

cancellation\_token.cancel()

try:

result = await run # This will raise a CancelledError.

except asyncio.CancelledError:

print("Task was cancelled.")

如果再次运行，则会抛出[CancelledError](https://docs.python.org/3/library/asyncio-exceptions.html#asyncio.CancelledError)异常。

[UserProxyAgent](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.UserProxyAgent).在team运行过程中用来提供用户反馈：

model\_client = OpenAIChatCompletionClient(model="gpt-4o-mini")

assistant = AssistantAgent("assistant", model\_client=model\_client)

user\_proxy = UserProxyAgent("user\_proxy", input\_func=input) # Use input() to get user input from console.

# Create the termination condition which will end the conversation when the user says "APPROVE".

termination = TextMentionTermination("APPROVE")

# Create the team.

team = RoundRobinGroupChat([assistant, user\_proxy], termination\_condition=termination)

# Run the conversation and stream to the console.

stream = team.run\_stream(task="Write a 4-line poem about the ocean.")

# Use asyncio.run(...) when running in a script.

await Console(stream)

以上例子只用用户输入APPROVE后task才结束。

# Termination

所有都继承自[TerminationCondition](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.base.html#autogen_agentchat.base.TerminationCondition)，一旦达到终止条件，会话结束。所有条件reset。主要有以下termination：

[MaxMessageTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.MaxMessageTermination): [TextMentionTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.TextMentionTermination) [TokenUsageTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.TokenUsageTermination) [TimeoutTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.TimeoutTermination) [HandoffTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.HandoffTermination)（当交给一个专门的目标时终止）

[SourceMatchTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.SourceMatchTermination): [ExternalTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.ExternalTermination): [StopMessageTermination](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.conditions.html#autogen_agentchat.conditions.StopMessageTermination):

# Custom Agents

所有custom agent必须继承[BaseChatAgent](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent)，并且实现以下抽象方法，

[on\_messages()](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_messages)定义了agent响应message的response的行为

[on\_reset()](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.on_reset)当agent调用reset方法时调用

[produced\_message\_types](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.BaseChatAgent.produced_message_types) 接受的[ChatMessage](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.messages.html#autogen_agentchat.messages.ChatMessage)类型列表

class CountDownAgent(BaseChatAgent):

def \_\_init\_\_(self, name: str, count: int = 3):

super().\_\_init\_\_(name, "A simple agent that counts down.")

self.\_count = count

@property

def produced\_message\_types(self) -> Sequence[type[ChatMessage]]:

return (TextMessage,)

async def on\_messages(self, messages: Sequence[ChatMessage], cancellation\_token: CancellationToken) -> Response:

# Calls the on\_messages\_stream.

response: Response | None = None

async for message in self.on\_messages\_stream(messages, cancellation\_token):

if isinstance(message, Response):

response = message

assert response is not None

return response

async def on\_messages\_stream(

self, messages: Sequence[ChatMessage], cancellation\_token: CancellationToken

) -> AsyncGenerator[AgentEvent | ChatMessage | Response, None]:

inner\_messages: List[AgentEvent | ChatMessage] = []

for i in range(self.\_count, 0, -1):

msg = TextMessage(content=f"{i}...", source=self.name)

inner\_messages.append(msg)

yield msg

# The response is returned at the end of the stream.

# It contains the final message and all the inner messages.

yield Response(chat\_message=TextMessage(content="Done!", source=self.name), inner\_messages=inner\_messages)

async def on\_reset(self, cancellation\_token: CancellationToken) -> None:

pass

# Managing State

agent\_state = await assistant\_agent.save\_state()

await new\_assistant\_agent.load\_state(agent\_state)

我们使用save\_state保存agent的state和load\_state加载对应的state

同时可以导出到本地和导入

## save state to disk

with open("coding/team\_state.json", "w") as f:

json.dump(team\_state, f)

## load state from disk

with open("coding/team\_state.json", "r") as f:

team\_state = json.load(f)

# [SelectorGroupChat](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.SelectorGroupChat)

[SelectorGroupChat](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.SelectorGroupChat)类似[RoundRobinGroupChat](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.RoundRobinGroupChat)，但是有一个基于模型的下一个发言者选择链路。当整个team通过run或runstream收到一个task，以下步骤开始：

1.团队分析当前的聊天上下文（包括历史聊天记录，与会者名字，描述），通过模型来决定下一个发言者。默认情况下，团队不会连续选择同一个发言者除非它是唯一一个可用的agent。这个设定可以通过设置allow\_repeated\_speaker=True来修改，你也可以重写提供自定义选择函数来完成。

2．团队提示所选的说话者代理提供响应，然后将其广播给所有其他参与者。

3.终止条件被用来检测是否会话结束，如果没有，重复第一步。

4.会话结束时，返回taskresult,包含本次任务的会话历史。

会话结束后，会话历史仍然保持在上下文中，可以通过reset方法清除。

# Swarm

让单个agent提供了一个交接任务的能力而不用通过中央编排器来控制。类似[SelectorGroupChat](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.SelectorGroupChat) 和[RoundRobinGroupChat](https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.teams.html#autogen_agentchat.teams.RoundRobinGroupChat)，各个agent广播它们各自的响应，这样所有的agent共享相同的上下文。不同之处在于，发言的agent是根据上下文中最新的Handoff消息选择的。每个在team中的agent都能通过handoffs参数产生交接信息，该参数可以配置下一个交接的agent。第一个发言代码处理完成后决定下一个发言代理，直至终止。

# Magentic-One

Magentic-One是一个通用的多agent系统，它用于处理各种领域的开放式网络和文件任务。

编排者首先创建一个解决任务的计划，在维护的任务分类账中收集所需的事实和有根据的猜测。在计划的每一步，编排者都会创建一个进度分类账，在其中自我反映任务进度并检查任务是否完成。如果任务尚未完成，它会为Magentic one其他代理之一分配一个子任务来完成。在分配的代理完成其子任务后，编排器会更新进度分类账，并以此方式继续，直到任务完成

Magentic-One由以下agent组成：

协调人：负责任务分解和规划的主要代理，指导其他代理执行子任务，跟踪总体进度，并根据需要采取纠正措施

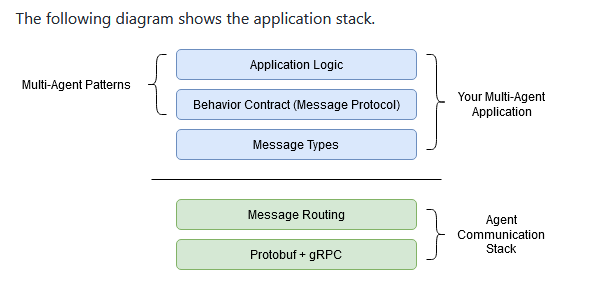
WebSurfer：这是一个基于LLM的代理，精通命令和管理基于Chromium的web浏览器的状态。对于每个传入的请求，WebSurfer在浏览器上执行一个操作，然后报告网页的新状态。WebSurfer的操作空间包括导航（例如访问URL、执行网络搜索）；网页操作（例如点击和打字）；以及阅读动作（例如总结或回答问题）。

FileSurfer：这是一个基于LLM的代理，它命令基于markdown的文件预览应用程序读取大多数类型的本地文件。FileSurfer还可以执行常见的导航任务，例如列出目录内容和导航文件夹结构

Coder：这是一个基于LLM的代理，通过其系统提示专门用于编写代码、分析从其他代理收集的信息或创建新工件

ComputerTerminal：最后，ComputerTerminal为团队提供了对控制台外壳的访问，在那里可以执行程序员的程序，也可以安装新的编程库。

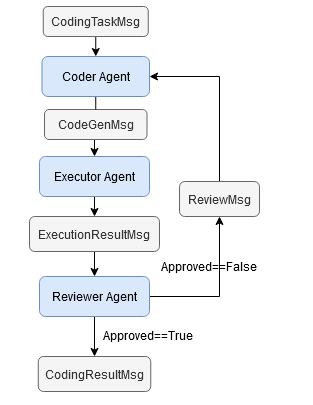
# Application Stack



堆栈的底部是基础消息传递和路由设施，使代理能够相互通信。

在堆栈的顶部，开发人员需要定义代理交换的消息类型。这组消息类型形成了代理必须遵守的行为契约，契约的实现决定了代理如何处理消息。行为契约有时也被称为消息协议。执行行为契约是开发者的责任。

以下是多agent的架构：



## Agent ID

Agentid是agent的唯一标识，包含两个组件：  
Agent ID = (Agent Type, Agent Key)

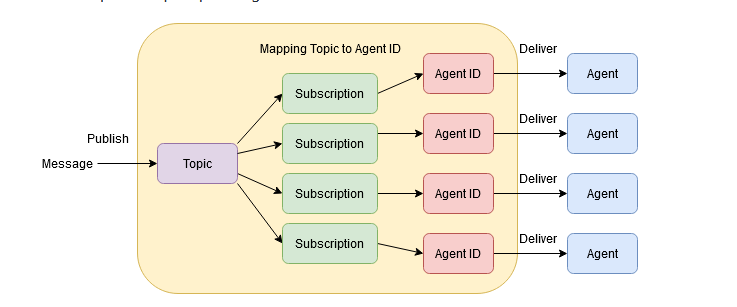
## Agent Lifecycle

当运行时向给定ID的代理实例传递消息时，它要么获取该实例，要么在不存在的情况下创建它。

## Topic

Topic由两个组件组成，Topic = (Topic Type, Topic Source)，Topic source是topic的唯一标识符，允许发布者限制消息的范围

## Subscription



上图显示了主题和订阅之间的关系。代理运行时跟踪订阅并使用它们向代理传递消息。

如果某个主题没有订阅，则发布到此主题的消息将不会传递给任何代理。如果一个主题有多个订阅，则消息将在所有订阅后仅传递给每个收件人代理一次。应用程序可以使用代理运行时的API添加或删除订阅。

# Type-based Subscriptions for single-tenant, single topic scenario

TypeSubscription(topic\_type="default", agent\_type="triage\_agent")

# The agent IDs created based on the topic source

AgentID("triage\_agent", "default")

## Implementing an Agent

要实现代理，开发人员必须子类化RoutedAgent类，并使用message\_handler装饰器为代理预期处理的每种消息类型实现一个消息处理程序方法。

@dataclass

class MyMessageType:

content: str

class MyAgent(RoutedAgent):

def \_\_init\_\_(self) -> None:

super().\_\_init\_\_("MyAgent")

@message\_handler

async def handle\_my\_message\_type(self, message: MyMessageType, ctx: MessageContext) -> None:

print(f"{self.id.type} received message: {message.content}")

Registering Agent Type

为了使代理对运行时可用，开发人员可以使用BaseAgent类的register（）类方法。注册过程将一个由字符串唯一标识的代理类型与一个创建给定类的代理类型实例的工厂函数相关联。

from autogen\_core import SingleThreadedAgentRuntime

runtime = SingleThreadedAgentRuntime()

await MyAgent.register(runtime, "my\_agent", lambda: MyAgent())

await MyAssistant.register(runtime, "my\_assistant", lambda: MyAssistant("my\_assistant"))

一旦注册了代理类型，我们就可以使用AgentId向代理实例发送直接消息。运行时将在第一次向该实例传递消息时创建该实例

runtime.start() # Start processing messages in the background.

await runtime.send\_message(MyMessageType("Hello, World!"), AgentId("my\_agent", "default"))

await runtime.send\_message(MyMessageType("Hello, World!"), AgentId("my\_assistant", "default"))

await runtime.stop() # Stop processing messages in the background.

LOG输出如下：

my\_agent received message: Hello, World!

my\_assistant received message: Hello, World!

my\_assistant responded: Hello! How can I assist you today?

# Message and Communication

Messages是由唯一用来跟agent通信的方式，通过继承BaseModel或者用@dataclass注解定义

Message里面只能定义数据，不含有其他逻辑性代码。

## Message Handlers

当一个代理在消息运行时收到一个消息时，它将调用 @message\_handler

装饰的消息处理方法，否则将抛出异常。

class MyAgent(RoutedAgent):

@message\_handler

async def on\_text\_message(self, message: TextMessage, ctx: MessageContext) -> None:

print(f"Hello, {message.source}, you said {message.content}!")

@message\_handler

async def on\_image\_message(self, message: ImageMessage, ctx: MessageContext) -> None:

print(f"Hello, {message.source}, you sent me {message.url}!")

接下来创建运行时agent，并注册agent类型

runtime = SingleThreadedAgentRuntime()

await MyAgent.register(runtime, "my\_agent", lambda: MyAgent("My Agent"))

测试如下

runtime.start()

agent\_id = AgentId("my\_agent", "default")

await runtime.send\_message(TextMessage(content="Hello, World!", source="User"), agent\_id)

await runtime.send\_message(ImageMessage(url="https://example.com/image.jpg", source="User"), agent\_id)

await runtime.stop\_when\_idle()

输出以下结果

Hello, User, you said Hello, World!!

Hello, User, you sent me https://example.com/image.jpg!

当发送第一条消息时，运行时MyAgent会自动实例化，AgentId("my\_agent", "default")

一对一发消息用send\_message，一对多用add\_subscription注册监听，

和publish\_message发送广播消息。@type\_subscription来装饰消息处理函数，[TopicId](https://microsoft.github.io/autogen/stable/reference/python/autogen_core.html#autogen_core.TopicId)来制定发送的目标。

@type\_subscription(topic\_type="default")

class ReceivingAgent(RoutedAgent):

@message\_handler

async def on\_my\_message(self, message: Message, ctx: MessageContext) -> None:

print(f"Received a message: {message.content}")

处理消息依然用@message\_handler

class BroadcastingAgent(RoutedAgent):

@message\_handler

async def on\_my\_message(self, message: Message, ctx: MessageContext) -> None:

await self.publish\_message(

Message("Publishing a message from broadcasting agent!"),

topic\_id=TopicId(type="default", source=self.id.key),

)

创建发送逻辑如下

runtime = SingleThreadedAgentRuntime()

# Option 1: with type\_subscription decorator

# The type\_subscription class decorator automatically adds a TypeSubscription to

# the runtime when the agent is registered.

await ReceivingAgent.register(runtime, "receiving\_agent", lambda: ReceivingAgent("Receiving Agent"))

# Option 2: with TypeSubscription

await BroadcastingAgent.register(runtime, "broadcasting\_agent", lambda: BroadcastingAgent("Broadcasting Agent"))

await runtime.add\_subscription(TypeSubscription(topic\_type="default", agent\_type="broadcasting\_agent"))

# Start the runtime and publish a message.

runtime.start()

await runtime.publish\_message(

Message("Hello, World! From the runtime!"), topic\_id=TopicId(type="default", source="default")

)

await runtime.stop\_when\_idle()

输出：Received a message: Hello, World! From the runtime!

Received a message: Publishing a message from broadcasting agent!

DefaultTopicId用于创建一个主题，该主题使用“默认”作为主题类型的默认值，并使用发布代理的key作为topic源的默认值。default\_subscription（）用于创建订阅默认主题的类型订阅。我们可以通过使用DefaultTopicId和default\_subscription（）来简化BroadcastingAgent。

@default\_subscription

class BroadcastingAgentDefaultTopic(RoutedAgent):

await runtime.publish\_message(Message("Hello, World! From the runtime!"), topic\_id=DefaultTopicId())

## Manage Model Context

Agent总是以一个新的上下文进行响应，该上下文仅包含系统消息和最新用户的消息。我们可以使用autogen\_core.model\_context中的模型上下文类来使代理“记住”之前的对话。模型上下文支持存储和检索聊天完成消息。它总是与模型客户端一起使用，以生成基于LLM的响应。

例如，BufferedChatCompletionContext是一个最近使用的（MRU）上下文，它存储了最新的buffer\_size消息数。这对于避免许多LLM中的上下文溢出非常有用。  
self.\_model\_context = BufferedChatCompletionContext(buffer\_size=5)

await self.\_model\_context.add\_message(AssistantMessage(content=response.content, source=self.metadata["type"]))

# Tools

[PythonCodeExecutionTool](https://microsoft.github.io/autogen/stable/reference/python/autogen_ext.tools.code_execution.html#autogen_ext.tools.code_execution.PythonCodeExecutionTool)允许agent执行python代码片段。

[DockerCommandLineCodeExecutor](https://microsoft.github.io/autogen/stable/reference/python/autogen_ext.code_executors.docker.html#autogen_ext.code_executors.docker.DockerCommandLineCodeExecutor)支持代码运行在docker环境。

[FunctionTool](https://microsoft.github.io/autogen/stable/reference/python/autogen_core.tools.html#autogen_core.tools.FunctionTool)可以用来自定义tools

async def get\_stock\_price(ticker: str, date: Annotated[str, "Date in YYYY/MM/DD"]) -> float:

# Returns a random stock price for demonstration purposes.

return random.uniform(10, 200)

# Create a function tool.

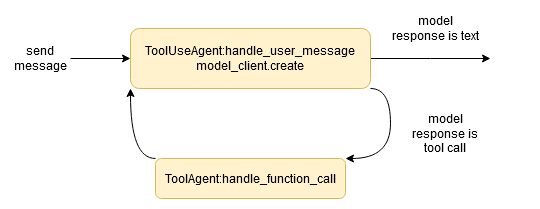
stock\_price\_tool = FunctionTool(get\_stock\_price, description="Get the stock price.")

# Run the tool.

cancellation\_token = CancellationToken()

result = await stock\_price\_tool.run\_json({"ticker": "AAPL", "date": "2021/01/01"}, cancellation\_token)

ToolUseAgent类使用便利函数tool\_agent\_caller\_loop（）来处理模型和工具代理之间的交互。核心思想可以用一个简单的控制流图来描述：



ToolUseAgent的handle\_user\_message处理程序处理来自用户的消息，并确定模型是否生成了工具调用。如果模型生成了工具调用，则处理程序向ToolAgent代理发送函数调用消息以执行工具，然后使用工具调用的结果再次查询模型。此过程一直持续到模型停止生成工具调用，此时最终响应将返回给用户。

messages = await tool\_agent\_caller\_loop(

self,

tool\_agent\_id=self.\_tool\_agent\_id,

model\_client=self.\_model\_client,

input\_messages=session,

tool\_schema=self.\_tool\_schema,

cancellation\_token=ctx.cancellation\_token,

)

## Single Message & Multiple Processors



## Multiple messages & Multiple Processors





使用await ClosureAgent.register\_closure将发送message汇总

## Message Protocol

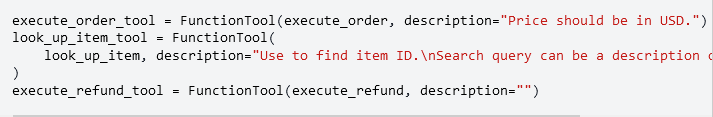
首先，用户或外部代理向所有参与者的公共主题发布GroupChatMessage消息。

群聊管理器选择下一个发言者，向该代理发送RequestToSpeak消息。

代理在收到RequestToSpeak消息后，向公共主题发布GroupChatMessage消息。

此过程将继续，直到群聊管理器达到终止条件，然后停止发出RequestToSpeak消息，群聊结束。

如果AI代理不需要将任务移交给其他代理，他们可以使用常规工具来完成任务。我们使用简单的函数定义工具，并使用FunctionTool包装器创建工具。



## Delegate tools for the AI agents

除了常规工具外，AI代理还可以使用称为委托工具的特殊工具将任务委托给其他代理。委托工具的概念仅在这种设计模式中使用，委托工具也被定义为简单的功能。在这种设计模式中，我们将委托工具与常规工具区分开来，因为当AI代理调用委托工具时，我们会将任务转移到另一个代理，而不是继续使用同一代理中的模型生成响应。

def transfer\_to\_sales\_agent() -> str:

return sales\_agent\_topic\_type

transfer\_to\_sales\_agent\_tool = FunctionTool(

transfer\_to\_sales\_agent, description="Use for anything sales or buying related."

)

# Multi-Agent Debate

此模式中有两种类型的代理：求解器代理和聚合器代理。工作代理负责解决问题并相互交换响应。聚合器代理负责将问题分发给工作代理，等待它们的最终响应，并聚合响应以获得最终答案。

该模式的工作原理如下：

用户向聚合器代理发送一个问题。

聚合器代理将问题分发给求解器代理。

每个工作器代理处理问题，并向其邻居发布响应。

每个工作器代理使用其邻居的响应来优化其响应，并发布新的响应。

重复步骤4进行固定轮数。在最后一轮中，每个工作代理都会发布最终响应。

聚合器代理使用多数投票来聚合所有求解器代理的最终响应，以获得最终答案，并发布答案。

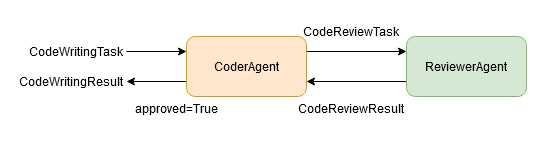
## Solver Agent

求解器代理负责解决问题，并与其他求解器代理交换响应。在收到SolverRequest后，求解器代理使用LLM生成答案。然后，它根据轮数发布IntermediateSolverResponse或FinalSolverResponse。

# Reflection

在AutoGen和代理的上下文中，反射可以实现为一对代理，其中第一个代理生成消息，第二个代理生成对消息的响应。两个代理继续交互，直到它们达到停止条件，例如最大迭代次数或第二个代理的批准。

让我们使用AutoGen代理实现一个简单的反射设计模式。将有两个代理：编码代理和审阅代理，编码代理将生成代码片段，审阅代理将生成对代码片段的评论。

、