Multi-agent Conversation Framework

AutoGen offers a unified multi-agent conversation framework as a high-level abstraction of using foundation models. It features capable, customizable and conversable agents which integrate LLMs, tools, and humans via automated agent chat. By automating chat among multiple capable agents, one can easily make them collectively perform tasks autonomously or with human feedback, including tasks that require using tools via code.

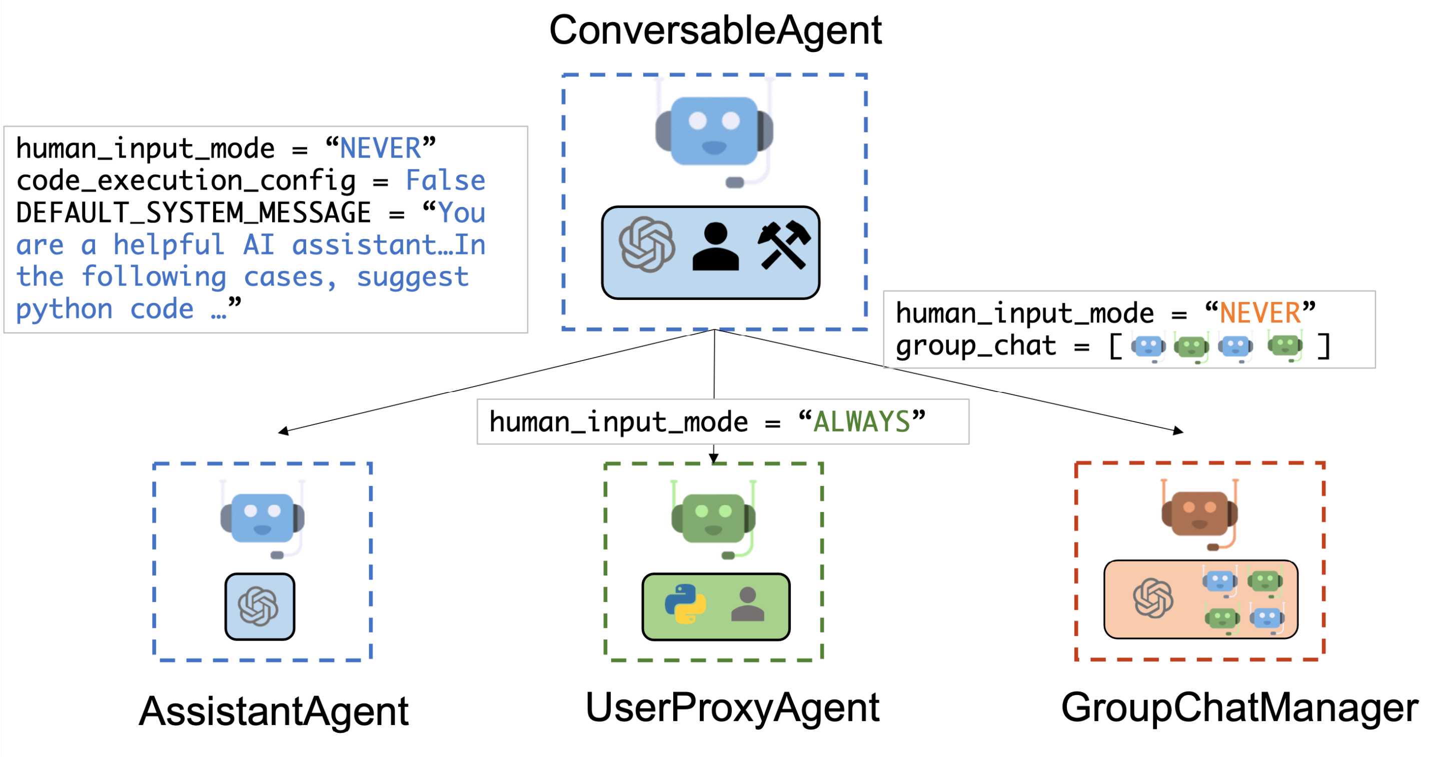
This framework simplifies the orchestration, automation and optimization of a complex LLM workflow. It maximizes the performance of LLM models and overcome their weaknesses. It enables building next-gen LLM applications based on multi-agent conversations with minimal effort.

Agents

AutoGen abstracts and implements conversable agents designed to solve tasks through inter-agent conversations. Specifically, the agents in AutoGen have the following notable features:

- Conversable: Agents in AutoGen are conversable, which means that any agent can send and receive messages from other agents to initiate or continue a conversation
- Customizable: Agents in AutoGen can be customized to integrate LLMs, humans, tools, or a combination of them.

The figure below shows the built-in agents in AutoGen.


ConversableAgent
human_input_mode = "NEVER"
code_execution_config = False
DEFAULT_SYSTEM_MESSAGE = "You are a helpful AI assistant…In the following cases, suggest python code …"
human_input_mode = "NEVER"
group_chat = [ ]
human_input_mode = "ALWAYS"
AssistantAgent
UserProxyAgent
GroupChatManager

We have designed a generic `ConversableAgent` class for Agents that are capable of conversing with each other through the exchange of messages to jointly finish a task. An agent can communicate with other agents and perform actions. Different agents can differ in what actions they perform after receiving messages. Two representative subclasses are `AssistantAgent` and `UserProxyAgent`

- The `AssistantAgent` is designed to act as an AI assistant, using LLMs by default but not requiring human input or code execution. It could write Python code (in a Python coding block) for a user to execute when a message (typically a description of a task that needs to be solved) is received. Under the hood, the Python code is written by LLM (e.g., GPT-4). It can also receive the execution results and suggest corrections or bug fixes. Its behavior can be altered by passing a new system message. The LLM inference configuration can be configured via [`llm_config`].

- The `UserProxyAgent` is conceptually a proxy agent for humans, soliciting human input as the agent's reply at each interaction turn by default and also having the capability to execute code and call functions or tools. The `UserProxyAgent` triggers code execution automatically when it detects an executable code block in the received message and no human user input is provided. Code execution can be disabled by setting the `code_execution_config` parameter to False. LLM-based response is disabled by default. It can be enabled by setting `llm_config` to a dict corresponding to the inference configuration. When `llm_config` is set as a dictionary, `UserProxyAgent` can generate replies using an LLM when code execution is not performed.

The auto-reply capability of `ConversableAgent` allows for more autonomous multi-agent communication while retaining the possibility of human intervention. One can also easily extend it by registering reply functions with the `register_reply()` method.

In the following code, we create an `AssistantAgent` named "assistant" to serve as the assistant and a `UserProxyAgent` named "user_proxy" to serve as a proxy for the human user. We will later employ these two agents to solve a task.

```
from autogen import AssistantAgent, UserProxyAgent

# create an AssistantAgent instance named "assistant"
assistant = AssistantAgent(name="assistant")

# create a UserProxyAgent instance named "user_proxy"
user_proxy = UserProxyAgent(name="user_proxy")
```

Tool calling

Tool calling enables agents to interact with external tools and APIs more efficiently. This feature allows the AI model to intelligently choose to output a JSON object containing arguments to call specific tools based on the user's input. A tool to be called is specified with a JSON schema describing its parameters and their types. Writing such JSON schema is complex and error-prone and that is why AutoGen framework provides two high level function decorators for automatically generating such schema using type hints on standard Python datatypes or Pydantic models:

1. `ConversableAgent.register_for_llm` is used to register the function as a Tool in the `llm_config` of a ConversableAgent. The ConversableAgent agent can propose execution of a registered Tool, but the actual execution will be performed by a UserProxy agent.

2. `ConversableAgent.register_for_execution` is used to register the function in the `function_map` of a UserProxy agent.

The following examples illustrates the process of registering a custom function for currency exchange calculation that uses type hints and standard Python datatypes:

1. First, we import necessary libraries and configure models using `autogen.config_list_from_json` function:

```
from typing import Literal

from pydantic import BaseModel, Field
from typing_extensions import Annotated

import autogen

config_list = autogen.config_list_from_json(
    "OAI_CONFIG_LIST",
    filter_dict={
        "model": ["gpt-4", "gpt-3.5-turbo", "gpt-3.5-turbo-16k"],
    },
)
```

2. We create an assistant agent and user proxy. The assistant will be responsible for suggesting which functions to call and the user proxy for the actual execution of a proposed function:

```
llm_config = {
    "config_list": config_list,
    "timeout": 120,
}

chatbot = autogen.AssistantAgent(
    name="chatbot",
    system_message="For currency exchange tasks, only use the functions you have been provided with. Reply TERMINAT
    llm_config=llm_config,
)

# create a UserProxyAgent instance named "user_proxy"
user_proxy = autogen.UserProxyAgent(
    name="user_proxy",
    is_termination_msg=lambda x: x.get("content", "") and x.get("content", "").rstrip().endswith("TERMINATE"),
    human_input_mode="NEVER",
    max_consecutive_auto_reply=10,
)
```

3. We define the function `currency_calculator` below as follows and decorate it with two decorators:
    - `@user_proxy.register_for_execution()` adding the function `currency_calculator` to `user_proxy.function_map`, and

- @chatbot.register_for_llm adding a generated JSON schema of the function to `llm_config` of `chatbot`.

```
CurrencySymbol = Literal["USD", "EUR"]


def exchange_rate(base_currency: CurrencySymbol, quote_currency: CurrencySymbol) -> float:
    if base_currency == quote_currency:
        return 1.0
    elif base_currency == "USD" and quote_currency == "EUR":
        return 1 / 1.1
    elif base_currency == "EUR" and quote_currency == "USD":
        return 1.1
    else:
        raise ValueError(f"Unknown currencies {base_currency}, {quote_currency}")

# NOTE: for Azure OpenAI, please use API version 2023-12-01-preview or later as
# support for earlier versions will be deprecated.
# For API versions 2023-10-01-preview or earlier you may
# need to set `api_style="function"` in the decorator if the default value does not work:
# `register_for_llm(description=..., api_style="function")`.
@user_proxy.register_for_execution()
@chatbot.register_for_llm(description="Currency exchange calculator.")
def currency_calculator(
    base_amount: Annotated[float, "Amount of currency in base_currency"],
    base_currency: Annotated[CurrencySymbol, "Base currency"] = "USD",
    quote_currency: Annotated[CurrencySymbol, "Quote currency"] = "EUR",
) -> str:
    quote_amount = exchange_rate(base_currency, quote_currency) * base_amount
    return f"{quote_amount} {quote_currency}"
```

Notice the use of Annotated to specify the type and the description of each parameter. The return value of the function must be either string or serializable to string using the `json.dumps()` or `Pydantic` model dump to JSON (both version 1.x and 2.x are supported).

You can check the JSON schema generated by the decorator `chatbot.llm_config["tools"]`:

```
[{'type': 'function', 'function':
 {'description': 'Currency exchange calculator.',
  'name': 'currency_calculator',
  'parameters': {'type': 'object',
   'properties': {'base_amount': {'type': 'number',
     'description': 'Amount of currency in base_currency'},
    'base_currency': {'enum': ['USD', 'EUR'],
     'type': 'string',
     'default': 'USD',
     'description': 'Base currency'},
    'quote_currency': {'enum': ['USD', 'EUR'],
     'type': 'string',
     'default': 'EUR',
     'description': 'Quote currency'}},
   'required': ['base_amount']}}}]
```

4. Agents can now use the function as follows:

```
user_proxy.initiate_chat(
    chatbot,
    message="How much is 123.45 USD in EUR?",
)
```

Output:

```
user_proxy (to chatbot):

How much is 123.45 USD in EUR?

--------------------------------------------------------------------------------
chatbot (to user_proxy):

***** Suggested tool Call: currency_calculator *****
Arguments:
{"base_amount":123.45,"base_currency":"USD","quote_currency":"EUR"}
********************************************************

--------------------------------------------------------------------------------

>>>>>>>> EXECUTING FUNCTION currency_calculator...
user_proxy (to chatbot):

***** Response from calling function "currency_calculator" *****
112.22727272727272 EUR
****************************************************************

--------------------------------------------------------------------------------
chatbot (to user_proxy):

123.45 USD is equivalent to approximately 112.23 EUR.
...

TERMINATE
```

Use of Pydantic models further simplifies writing of such functions. Pydantic models can be used for both the parameters of a function and for its return type. Parameters of such functions will be constructed from JSON provided by an AI model, while the output will be serialized as JSON encoded string automatically.

The following example shows how we could rewrite our currency exchange calculator example:

```python
# defines a Pydantic model
class Currency(BaseModel):
  # parameter of type CurrencySymbol
  currency: Annotated[CurrencySymbol, Field(..., description="Currency symbol")]
  # parameter of type float, must be greater or equal to 0 with default value 0
  amount: Annotated[float, Field(0, description="Amount of currency", ge=0)]

@user_proxy.register_for_execution()
@chatbot.register_for_llm(description="Currency exchange calculator.")
def currency_calculator(
  base: Annotated[Currency, "Base currency: amount and currency symbol"],
  quote_currency: Annotated[CurrencySymbol, "Quote currency symbol"] = "USD",
) -> Currency:
  quote_amount = exchange_rate(base.currency, quote_currency) * base.amount
  return Currency(amount=quote_amount, currency=quote_currency)
```

The generated JSON schema has additional properties such as minimum value encoded:

```python
[{'type': 'function', 'function':
 {'description': 'Currency exchange calculator.',
  'name': 'currency_calculator',
  'parameters': {'type': 'object',
   'properties': {'base': {'properties': {'currency': {'description': 'Currency symbol',
       'enum': ['USD', 'EUR'],
       'title': 'Currency',
       'type': 'string'},
      'amount': {'default': 0,
       'description': 'Amount of currency',
       'minimum': 0.0,
       'title': 'Amount',
       'type': 'number'}},
     'required': ['currency'],
     'title': 'Currency',
     'type': 'object',
     'description': 'Base currency: amount and currency symbol'},
    'quote_currency': {'enum': ['USD', 'EUR'],
     'type': 'string',
     'default': 'USD',
     'description': 'Quote currency symbol'}},
   'required': ['base']}}}]
```

For more in-depth examples, please check the following:

- Currency calculator examples - View Notebook
- Use Provided Tools as Functions - View Notebook
- Use Tools via Sync and Async Function Calling - View Notebook

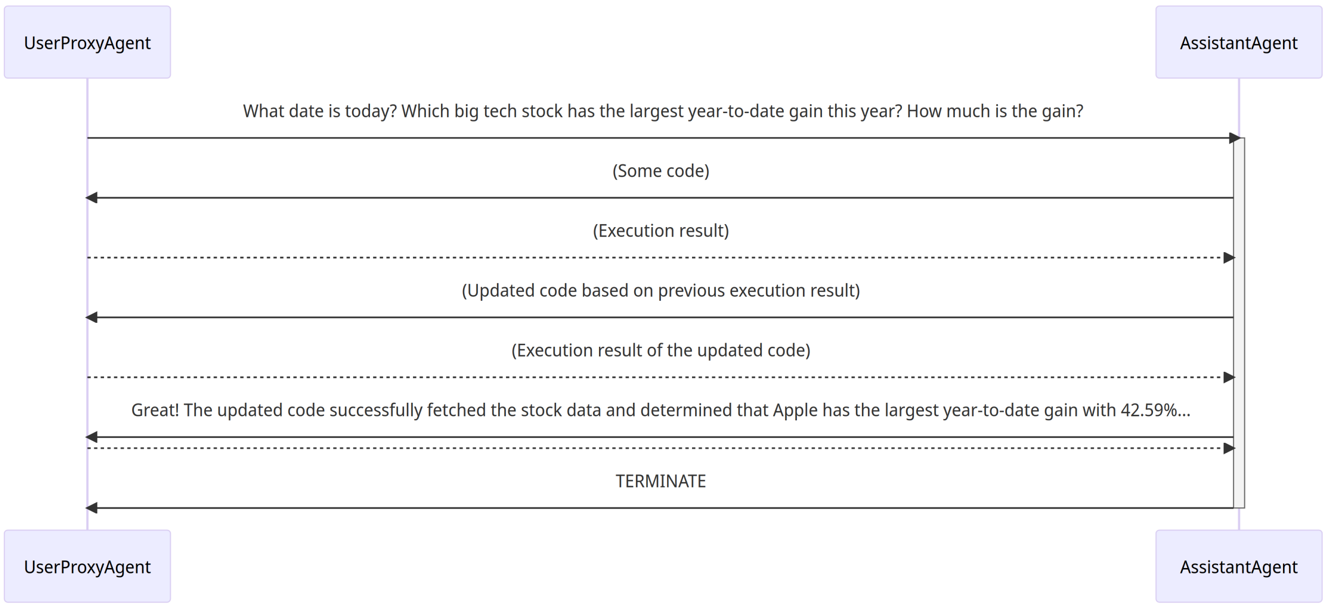
Multi-agent Conversations

A Basic Two-Agent Conversation Example

Once the participating agents are constructed properly, one can start a multi-agent conversation session by an initialization step as shown in the following code:

```
# the assistant receives a message from the user, which contains the task description
user_proxy.initiate_chat(
    assistant,
    message="""What date is today? Which big tech stock has the largest year-to-date gain this year? How much is th
)
```

After the initialization step, the conversation could proceed automatically. Find a visual illustration of how the user_proxy and assistant collaboratively solve the above task autonomously below:



What date is today? Which big tech stock has the largest year-to-date gain this year? How much is the gain?
(Some code)
(Execution result)
(Updated code based on previous execution result)
(Execution result of the updated code)
Great! The updated code successfully fetched the stock data and determined that Apple has the largest year-to-date gain with 42.59%...
TERMINATE

1. The assistant receives a message from the user_proxy, which contains the task description.
2. The assistant then tries to write Python code to solve the task and sends the response to the user_proxy.
3. Once the user_proxy receives a response from the assistant, it tries to reply by either soliciting human input or preparing an automatically generated reply. If no human input is provided, the user_proxy executes the code and uses the result as the auto-reply.
4. The assistant then generates a further response for the user_proxy. The user_proxy can then decide whether to terminate the conversation. If not, steps 3 and 4 are repeated.

Supporting Diverse Conversation Patterns

Conversations with different levels of autonomy, and human-involvement patterns

On the one hand, one can achieve fully autonomous conversations after an initialization step. On the other hand, AutoGen can be used to implement human-in-the-loop problem-solving by configuring human involvement levels and patterns (e.g., setting the `human_input_mode` to `ALWAYS`), as human involvement is expected and/or desired in many applications.

Static and dynamic conversations

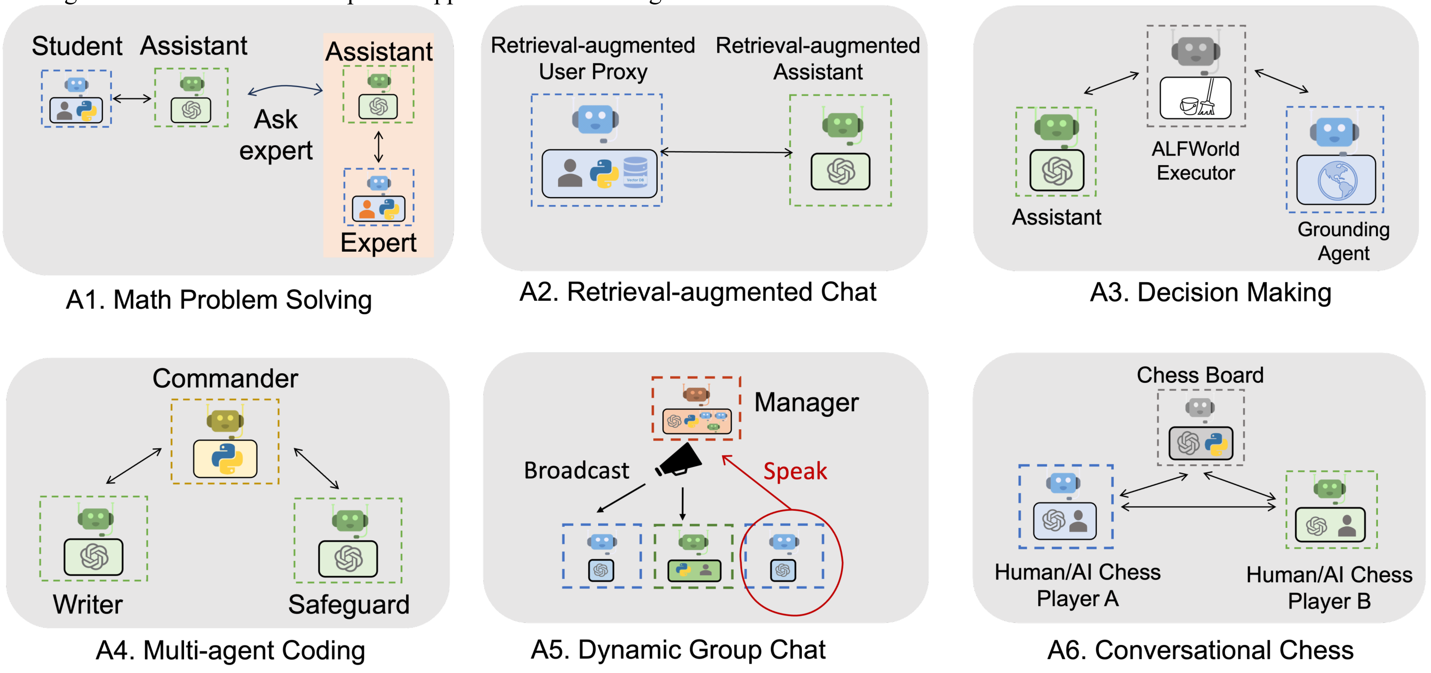
By adopting the conversation-driven control with both programming language and natural language, AutoGen inherently allows dynamic conversation. Dynamic conversation allows the agent topology to change depending on the actual flow of conversation under different input problem instances, while the flow of a static conversation always follows a pre-defined topology. The dynamic conversation pattern is useful in complex applications where the patterns of interaction cannot be predetermined in advance. AutoGen provides two general approaches to achieving dynamic conversation:

- Registered auto-reply. With the pluggable auto-reply function, one can choose to invoke conversations with other agents depending on the content of the current message and context. A working system demonstrating this type of dynamic conversation can be found in this code example, demonstrating a dynamic group chat. In the system, we register an auto-reply function in the group chat manager, which lets LLM decide who the next speaker will be in a group chat setting.

- LLM-based function call. In this approach, LLM decides whether or not to call a particular function depending on the conversation status in each inference call. By messaging additional agents in the called functions, the LLM can drive dynamic multi-agent conversation. A working system showcasing this type of dynamic conversation can be found in the multi-user math problem solving scenario, where a student assistant would automatically resort to an expert using function calls.

Diverse Applications Implemented with AutoGen

The figure below shows six examples of applications built using AutoGen.



Find a list of examples in this page: Automated Agent Chat Examples

For Further Reading

Interested in the research that leads to this package? Please check the following papers.

- AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework . Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang and Chi Wang. ArXiv 2023.
- An Empirical Study on Challenging Math Problem Solving with GPT-4. Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, Chi Wang. ArXiv preprint arXiv:2306.01337 (2023).

Edit this page

Previous
« Installation

Next
Enhanced Inference »