Text file content

Getting Started

AutoGen is a framework that enables development of LLM applications using multiple agents that can converse with each other to solve tasks. AutoGen agents are customizable, conversable, and seamlessly allow human participation. They can operate in various modes that employ combinations of LLMs, human inputs, and tools.

Main Features

- AutoGen enables building next-gen LLM applications based on multi-agent conversations with minimal effort. It simplifies the orchestration, automation, and optimization of a complex LLM workflow. It maximizes the performance of LLM models and overcomes their weaknesses.
- It supports diverse conversation patterns for complex workflows. With customizable and conversable agents, developers can use AutoGen to build a wide range of conversation patterns concerning conversation autonomy, the number of agents, and agent conversation topology.
- It provides a collection of working systems with different complexities. These systems span a wide range of applications from various domains and complexities. This demonstrates how AutoGen can easily support diverse conversation patterns.
- AutoGen provides enhanced LLM inference. It offers utilities like API unification and caching, and advanced usage patterns, such as error handling, multi-config inference, context programming, etc.

AutoGen is powered by collaborative research studies from Microsoft, Penn State University, and University of Washington.

Quickstart

Install from pip:

pip install pyautogen. Find more options in Installation.

For code execution, we strongly recommend installing the python docker package, and using docker.

Multi-Agent Conversation Framework

Autogen enables the next-gen LLM applications with a generic multi-agent conversation framework. It offers customizable and conversable agents which integrate LLMs, tools, and humans. 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. For example,

```
from autogen import AssistantAgent, UserProxyAgent, config_list_from_json
# Load LLM inference endpoints from an env variable or a file
# See https://microsoft.github.io/autogen/docs/FAQ#set-your-api-endpoints
# and OAI_CONFIG_LIST_sample.json
config_list = config_list_from_json(env_or_file="OAI_CONFIG_LIST")
assistant = AssistantAgent("assistant", llm_config={"config_list": config_list})
user_proxy = UserProxyAgent("user_proxy", code_execution_config={"work_dir": "coding"})
user_proxy.initiate_chat(assistant, message="Plot a chart of NVDA and TESLA stock price change YTD.")
# This initiates an automated chat between the two agents to solve the task
```

The figure below shows an example conversation flow with AutoGen.

Enhanced LLM Inferences

Autogen also helps maximize the utility out of the expensive LLMs such as ChatGPT and GPT-4. It offers enhanced LLM inference with powerful functionalities like tuning, caching, error handling, templating. For example, you can optimize generations by LLM with your own tuning data, success metrics and budgets.

```
# perform tuning for openai<1
config, analysis = autogen.Completion.tune(
data=tune_data,
metric="success",
mode="max",
eval_func=eval_func,
inference_budget=0.05,
optimization_budget=3,
num_samples=-1,
)
# perform inference for a test instance
response = autogen.Completion.create(context=test_instance, **config)
```

Where to Go Next ?

- Understand the use cases for multi-agent conversation and enhanced LLM inference.
- Find code examples.
- Read SDK.

- Learn about research around AutoGen.
- Roadmap
- Chat on Discord.
- Follow on Twitter.

If you like our project, please give it a star on GitHub. If you are interested in contributing, please read Contributor's Guide.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.

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

AssistantAgentis 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

UserProxyAgentis 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. The

UserProxyAgenttriggers 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_configparameter to False. LLM-based response is disabled by default. It can be enabled by setting

llm_configto a dict corresponding to the inference configuration. When

llm_configis set as a dictionary,

UserProxyAgentcan 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")
```

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 the gain?""",
)
```
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:
- The assistant receives a message from the user_proxy, which contains the task description.
- The assistant then tries to write Python code to solve the task and sends the response to the user_proxy.
- 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.
- 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).Examples
Automated Multi Agent Chat
AutoGen offers conversable agents powered by LLM, tool or human, which can be used to perform tasks collectively via automated chat. This framework allows tool use and human participation via multi-agent conversation. Please find documentation about this feature here.
Links to notebook examples:

Code Generation, Execution, and Debugging
- Automated Task Solving with Code Generation, Execution & Debugging - View Notebook
- Automated Code Generation and Question Answering with Retrieval Augmented Agents - View Notebook
- Automated Code Generation and Question Answering with Qdrant based Retrieval Augmented Agents - View Notebook
Multi-Agent Collaboration (>3 Agents)
- Automated Task Solving by Group Chat (with 3 group member agents and 1 manager agent) - View Notebook
- Automated Data Visualization by Group Chat (with 3 group member agents and 1 manager agent) - View Notebook
- Automated Complex Task Solving by Group Chat (with 6 group member agents and 1 manager agent) - View Notebook
- Automated Task Solving with Coding & Planning Agents - View Notebook
- Automated Task Solving with agents divided into 2 groups - View Notebook
- Automated Task Solving with transition paths specified in a graph - View Notebook
Applications
- Automated Chess Game Playing & Chitchatting by GPT-4 Agents - View Notebook
- Automated Continual Learning from New Data - View Notebook
- OptiGuide - Coding, Tool Using, Safeguarding & Question Answering for Supply Chain Optimization
Tool Use
- Web Search: Solve Tasks Requiring Web Info - View Notebook
- Use Provided Tools as Functions - View Notebook
- Use Tools via Sync and Async Function Calling - View Notebook
- Task Solving with Langchain Provided Tools as Functions - View Notebook
- RAG: Group Chat with Retrieval Augmented Generation (with 5 group member agents and 1 manager agent) - View Notebook
- Function Inception: Enable AutoGen agents to update/remove functions during conversations. - View Notebook
- Agent Chat with Whisper - View Notebook
- Constrained Responses via Guidance - View Notebook
Human Involvement
- Simple example in ChatGPT style View example
- Auto Code Generation, Execution, Debugging and Human Feedback - View Notebook
- Automated Task Solving with GPT-4 + Multiple Human Users - View Notebook
- Agent Chat with Async Human Inputs - View Notebook
Agent Teaching and Learning
- Teach Agents New Skills & Reuse via Automated Chat - View Notebook
- Teach Agents New Facts, User Preferences and Skills Beyond Coding - View Notebook
Multi-Agent Chat with OpenAI Assistants in the loop
- Hello-World Chat with OpenAi Assistant in AutoGen - View Notebook
- Chat with OpenAI Assistant using Function Call - View Notebook
- Chat with OpenAI Assistant with Code Interpreter - View Notebook
- Chat with OpenAI Assistant with Retrieval Augmentation - View Notebook
- OpenAI Assistant in a Group Chat - View Notebook
Multimodal Agent
- Multimodal Agent Chat with DALLE and GPT-4V - View Notebook
- Multimodal Agent Chat with Llava - View Notebook
- Multimodal Agent Chat with GPT-4V - View Notebook
Long Context Handling
- Conversations with Chat History Compression Enabled - View Notebook
Evaluation and Assessment
- AgentEval: A Multi-Agent System for Assess Utility of LLM-powered Applications - View Notebook
Automatic Agent Building
- Automatically Build Multi-agent System with AgentBuilder - View Notebook
Enhanced Inferences
Utilities
- API Unification - View Documentation with Code Example
- Utility Functions to Help Managing API configurations effectively - View Notebook
- Cost Calculation - View Notebook
Inference Hyperparameters Tuning
AutoGen offers a cost-effective hyperparameter optimization technique EcoOptiGen for tuning Large Language

Models. The research study finds that tuning hyperparameters can significantly improve the utility of them. Please find documentation about this feature here.
Links to notebook examples:Installation
Setup Virtual Environment
When not using a docker container, we recommend using a virtual environment to install AutoGen. This will ensure that the dependencies for AutoGen are isolated from the rest of your system.
Option 1: venv
You can create a virtual environment with
venv as below:
python3 -m venv pyautogen
source pyautogen/bin/activate
The following command will deactivate the current
venv environment:
deactivate
Option 2: conda
Another option is with
Conda, Conda works better at solving dependency conflicts than pip. You can install it by following this doc, and then create a virtual environment as below:
conda create -n pyautogen python=3.10 # python 3.10 is recommended as it's stable and not too old
conda activate pyautogen
The following command will deactivate the current
conda environment:
conda deactivate
Now, you're ready to install AutoGen in the virtual environment you've just created.
Python
AutoGen requires Python version >= 3.8, < 3.12. It can be installed from pip:
pip install pyautogen
pyautogen<0.2 requires
openai<1. Starting from pyautogen v0.2,
openai>=1 is required.
Migration guide to v0.2
openai v1 is a total rewrite of the library with many breaking changes. For example, the inference requires instantiating a client, instead of using a global class method.
Therefore, some changes are required for users of
pyautogen<0.2.
api_base->
base_url,
request_timeout->
timeoutin
llm_configand
config_list.
max_retry_periodand
retry_wait_timeare deprecated.
max_retriescan be set for each client.
- MathChat is unsupported until it is tested in future release.
autogen.Completionand
autogen.ChatCompletionare deprecated. The essential functionalities are moved to
autogen.OpenAIWrapper:
from autogen import OpenAIWrapper
client = OpenAIWrapper(config_list=config_list)
response = client.create(messages=[{"role": "user", "content": "2+2="}])
print(client.extract_text_or_completion_object(response))
- Inference parameter tuning and inference logging features are currently unavailable in
OpenAIWrapper. Logging will be added in a future release. Inference parameter tuning can be done via flaml.tune.
seedin autogen is renamed into
cache_seedto accommodate the newly added
seedparam in openai chat completion api.
use_cacheis removed as a kwarg in

OpenAIWrapper.create()for being automatically decided by
cache_seed: int | None. The difference between autogen's
cache_seedand openai's
seedis that:
- autogen uses local disk cache to guarantee the exactly same output is produced for the same input and when cache is hit, no openai api call will be made.
- openai's
seedis a best-effort deterministic sampling with no guarantee of determinism. When using openai's
seedwith
cache_seedset to None, even for the same input, an openai api call will be made and there is no guarantee for getting exactly the same output.
Optional Dependencies
docker
For the best user experience and seamless code execution, we highly recommend using Docker with AutoGen. Docker is a containerization platform that simplifies the setup and execution of your code. Developing in a docker container, such as GitHub Codespace, also makes the development convenient.
When running AutoGen out of a docker container, to use docker for code execution, you also need to install the python package
docker:
pip install docker
blendsearch
pyautogen<0.2 offers a cost-effective hyperparameter optimization technique EcoOptiGen for tuning Large Language Models. Please install with the [blendsearch] option to use it.
pip install "pyautogen[blendsearch]<0.2"
Example notebooks:
retrievechat
pyautogen supports retrieval-augmented generation tasks such as question answering and code generation with RAG agents. Please install with the [retrievechat] option to use it.
pip install "pyautogen[retrievechat]"
RetrieveChat can handle various types of documents. By default, it can process
plain text and PDF files, including formats such as 'txt', 'json', 'csv', 'tsv',
'md', 'html', 'htm', 'rtf', 'rst', 'jsonl', 'log', 'xml', 'yaml', 'yml' and 'pdf'.
If you install unstructured
(
pip install "unstructured[all-docs]"), additional document types such as 'docx',
'doc', 'odt', 'pptx', 'ppt', 'xlsx', 'eml', 'msg', 'epub' will also be supported.
You can find a list of all supported document types by using
autogen.retrieve_utils.TEXT_FORMATS.
Example notebooks:
Automated Code Generation and Question Answering with Retrieval Augmented Agents
Group Chat with Retrieval Augmented Generation (with 5 group member agents and 1 manager agent)
Automated Code Generation and Question Answering with Qdrant based Retrieval Augmented Agents
TeachableAgent
To use TeachableAgent, please install AutoGen with the [teachable] option.
pip install "pyautogen[teachable]"
Example notebook: Chatting with TeachableAgent
Large Multimodal Model (LMM) Agents
We offered Multimodal Conversable Agent and LLaVA Agent. Please install with the [lmm] option to use it.
pip install "pyautogen[lmm]"
Example notebooks:
mathchat
pyautogen<0.2 offers an experimental agent for math problem solving. Please install with the [mathchat] option to use it.
pip install "pyautogen[mathchat]<0.2"

My Current code we will be improving:

```
####
# To install required packages:
# pip install panel openai==1.3.6 panel==1.3.4
# pip install git+https://github.com/microsoft/autogen.git
# pip install python-dotenv

import autogen
import panel as pn
import openai
import os
import time
import asyncio
from autogen import config_list_from_json
from autogen.agentchat.contrib.gpt_assistant_agent import GPTAssistantAgent
from openai import OpenAI
from dotenv import load_dotenv

load_dotenv() # take environment variables from .env.

os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
assistant_id = os.environ.get("ASSISTANT_ID", None)
document = ''
client = OpenAI()
config_list = [
{
'model': 'gpt-4-1106-preview',
}
]
llm_config = {
"config_list": config_list,
"seed": 36,
"assistant_id": assistant_id,
"tools": [
{
"type": "retrieval"
}
],
"file_ids": [],
}

input_future = None

class MyConversableAgent(autogen.ConversableAgent):

async def a_get_human_input(self, prompt: str) -> str:
global input_future
chat_interface.send(prompt, user="System", respond=False)
# Create a new Future object for this input operation if none exists
if input_future is None or input_future.done():
input_future = asyncio.Future()

# Wait for the callback to set a result on the future
await input_future

# Once the result is set, extract the value and reset the future for the next input operation
input_value = input_future.result()
input_future = None
return input_value
```

```
user_proxy = MyConversableAgent(name="user_proxy",
code_execution_config=False,
is_termination_msg=lambda msg: "TERMINATE" in msg["content"],
human_input_mode="ALWAYS")

gpt_assistant = GPTAssistantAgent(name="assistant",
instructions="You are adapt at question answering",
llm_config=llm_config)

avatar = {user_proxy.name:"■■■", gpt_assistant.name:"■"}

def print_messages(recipient, messages, sender, config):

print(f"Messages from: {sender.name} sent to: {recipient.name} | num messages: {len(messages)} | message:
{messages[-1]}")
chat_interface.send(messages[-1]['content'], user=sender.name, avatar=avatar[sender.name], respond=False)

return False, None # required to ensure the agent communication flow continues

user_proxy.register_reply(
[autogen.Agent, None],
reply_func=print_messages,
config={"callback": None},
)
gpt_assistant.register_reply(
[autogen.Agent, None],
reply_func=print_messages,
config={"callback": None},
)

initiate_chat_task_created = False

async def delayed_initiate_chat(agent, recipient, message):

global initiate_chat_task_created
# Indicate that the task has been created
initiate_chat_task_created = True

await asyncio.sleep(2)

# Now initiate the chat
await agent.a_initiate_chat(recipient, message=message)

recipient.delete_assistant()

if llm_config['file_ids'][0]:
client.files.delete(llm_config['file_ids'][0])
print(f"Deleted file with ID: {llm_config['file_ids'][0]}")

time.sleep(5)


async def callback(contents: str, user: str, instance: pn.chat.ChatInterface):

global initiate_chat_task_created
global input_future
global gpt_assistant
```

```
if not initiate_chat_task_created:
asyncio.create_task(delayed_initiate_chat(user_proxy, gpt_assistant, contents))

else:
if input_future and not input_future.done():
input_future.set_result(contents)
else:
print("There is currently no input being awaited.")

pn.extension(design="material")

chat_interface = pn.chat.ChatInterface(
callback=callback,

show_button_name=False,
sizing_mode="stretch_both",
min_height=600,
)

chat_interface.send("Ask your question about the document!!", user="System", respond=False)

uploading = pn.indicators.LoadingSpinner(value=False, size=50, name='No document')
file_input = pn.widgets.FileInput(name="PDF File", accept=".pdf")
text_area = pn.widgets.TextAreaInput(name='File Info', sizing_mode='stretch_both', min_height=600)

def file_callback(*events):

for event in events:
if event.name == 'filename':
file_name = event.new
if event.name == 'value':
file_content = event.new

uploading.value = True
uploading.name = 'Uploading'
file_path = file_name

with open(file_path, 'wb') as f:
f.write(file_content)

response = client.files.create(file=open(file_path, 'rb'), purpose='assistants')

found = False
while not found:
for file in all_files.data:
if file.id == response.id:
found = True
print(f"Uploaded file with ID: {response.id}\n {file}")

global gpt_assistant
llm_config['file_ids'] = [file.id]
gpt_assistant.delete_assistant()
gpt_assistant = GPTAssistantAgent(name="assistant",
instructions="You are adept at question answering",
llm_config=llm_config)
gpt_assistant.register_reply(
[autogen.Agent, None],
```

```
reply_func=print_messages,
config={"callback": None},
)

text_area.value = str(client.files.retrieve(file.id))

uploading.value = False
uploading.name = f"Document uploaded - {file_name}"
break
if not found:
time.sleep(5)
all_files = client.files.list()

# Set up a callback on file input value changes
file_input.param.watch(file_callback, ['value', 'filename'])

title = '## Please upload your document for RAG'
file_app = pn.Column(pn.pane.Markdown(title), file_input, uploading, text_area, sizing_mode='stretch_width',
min_height=500)

pn.template.FastListTemplate(
title="■AutoGen w/ RAG",
header_background="#2F4F4F",
accent_base_color="#2F4F4F",
main=[
chat_interface
],
sidebar=[file_app],
sidebar_width=400,
).servable()
### End of my code
```