Development Platforms for Agentic Software

Gustavo Oliva

www.gaoliva.com

How to cite this session?

```
@misc{Oliva2024AIwareTutorial,
author = {Gustavo Oliva},
title = {Development Platforms for Agentic Software},
howpublished = {Tutorial presented at the AIware Leadership Bootcamp 2024},
month = {November},
year = {2024},
address = {Toronto, Canada},
note = {Part of the AIware Leadership Bootcamp series.},
url = {https://aiwarebootcamp.io/slides/2024_aiwarebootcamp_oliva_development_platforms_for_agentic_software.pdf}}
```

Overview of the session

- ❑ Introduction (5min)
- ❑ Deep dive into Microsoft AutoGen with examples (50min)
- ❑ Creating an AutoGen playground for experimentation (10min)
- ❑ Other agentic development platforms (5 min)
- ❑ Standardization efforts for FM-powered Agents (5 min)
- ❑ Beyond this presentation (1 min)

Overview of the session

- ❑ **Introduction**
- ❑ Deep dive into Microsoft AutoGen with examples
- ❑ Creating an AutoGen playground for experimentation
- ❑ Other agentic development platforms
- ❑ Standardization efforts for FM-powered Agents
- ❑ Beyond this presentation

Introduction

- The idea of **multi-agent systems** have been around for a while (e.g., agents in Reinforcement Learning)
- Multi-agent systems are typically designed around **autonomous agents that interact with each other to achieve a broader goal** (e.g., fix a software bug)
  - Suitable architecture for complex problems that require decomposition (each agent focuses on solving one part of the problem)
- With the advent of foundation models (particularly smarter LLMs), the community quickly saw the potential of creating **FM-powered agents** and **FM-powered multi-agent systems**.
- Shortly after, the need for **flexible multi-agent frameworks and platforms** emerged…

[…] *AGI will take the form factor of some kind of an **AI agent**. And it's **not just going to be a single agent.*** [1]

Andrej Karpathy (former director of AI and Autopilot Vision at Tesla. Now with OpenAI)

[1] https://www.youtube.com/watch?v=aGV3aycnwhA

The need for multi-agent frameworks

- **AutoGen** [1] from Microsoft became one of the most popular multi-agent frameworks in late 2023
- Multi-agent systems from that time (e.g., BabyAGI [1], MetaGPT [2]) served as inspiration to define the key set of features in AutoGen
  - A **reusable** framework
  - Flexible **agent conversation patterns**
  - **Code execution** capability
  - **Human-in-the-loop**

AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation

Qingyun Wu[†], Gagan Bansal[*], Jieyu Zhang[±], Yiran Wu[†], Beibin Li[*]

Erkang Zhu[*], Li Jiang[*], Xiaoyun Zhang[*], Shaokun Zhang[†], Jiale Liu[∓]

Ahmed Awadallah[*], Ryen W. White[*], Doug Burger[*], Chi Wang[*1]

[*]Microsoft Research, [†]Pennsylvania State University

[±]University of Washington, [∓]Xidian University

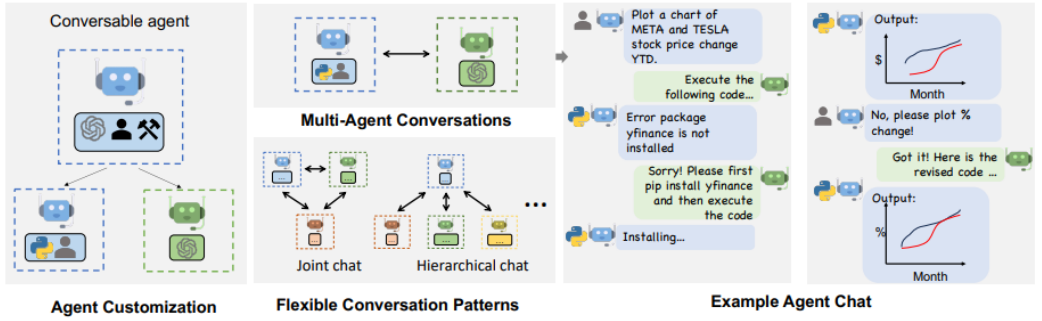


Figure 1: AutoGen enables diverse LLM-based applications using multi-agent conversations. (Left) AutoGen agents are conversable, customizable, and can be based on LLMs, tools, humans, or even a combination of them. (Top-middle) Agents can converse to solve tasks. (Right) They can form a chat, potentially with humans in the loop. (Bottom-middle) The framework supports flexible conversation patterns.

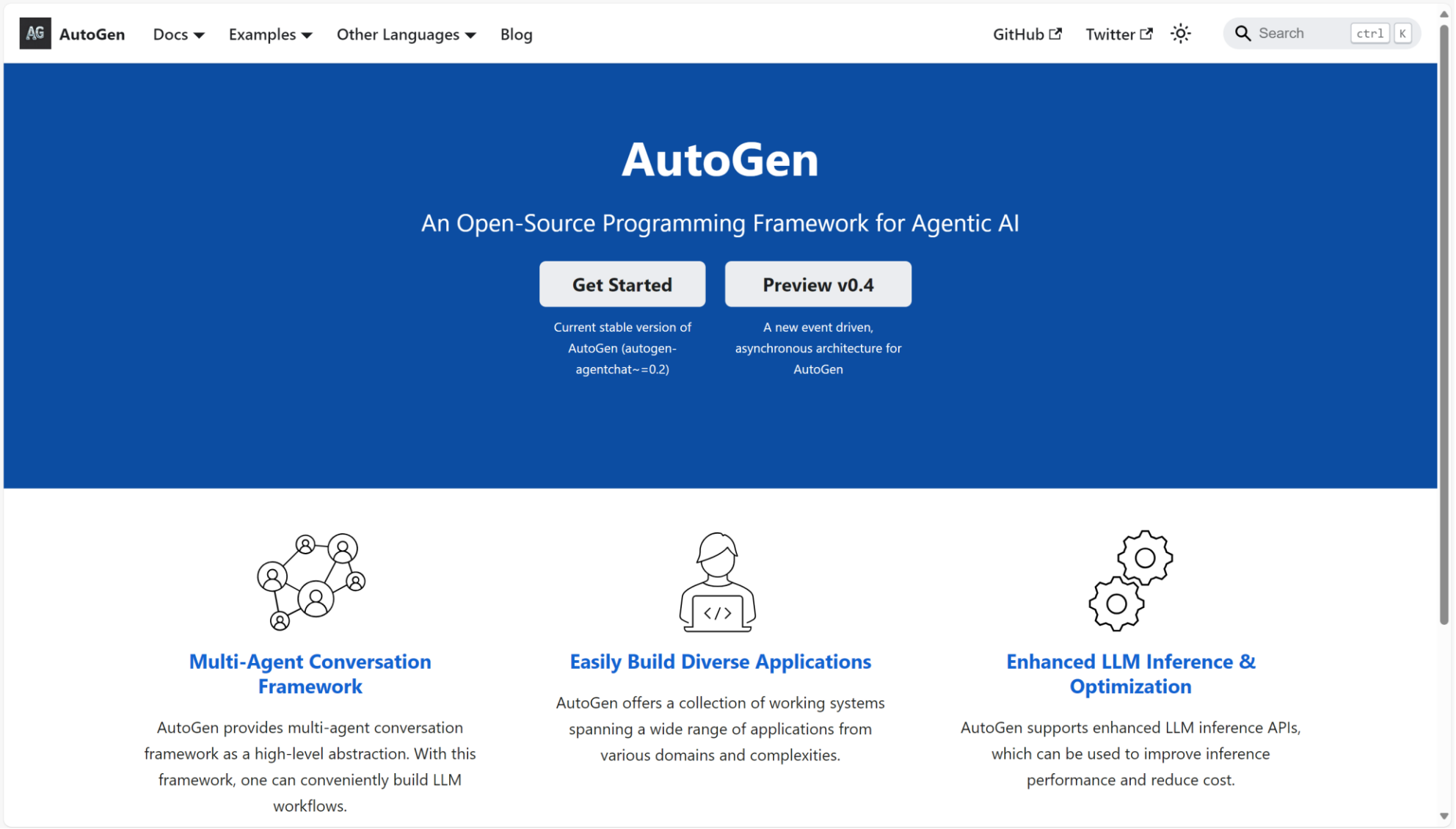
[1] Qingyun Wu et al. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155, 2023.
[2] BabyAGI. Github — babyagi. https://github.com/yoheinakajima/babyagi, 2023
[3] Hong et al. Metagpt: Meta programming for multi-agent collaborative framework. arXiv:2308.00352, 2023.

Cognitive architecture as the key value of an agentic system

- FMs have become more powerful over time and will continue to improve (e.g., reasoning capabilities with o1)
- These models are available to everyone building multi-agent systems
- **The key value (IP) of an agentic system thus lies in its cognitive architecture and not exactly in the models** themselves
  - How many agents should my system have?
  - What roles should they play?
  - What model should each agent use?
  - How should agents communicate?
  - How much human intervention should be prescribed?
  - How to best manage (the different types of) memory?
- And those are fundamentally an **SE problem!** (an open problem btw)

Overview of the session

- ❑ Introduction
- ❑ **Deep dive into Microsoft AutoGen with examples**
- ❑ Creating an AutoGen playground for experimentation
- ❑ Other agentic development platforms
- ❑ Standardization efforts for FM-powered Agents
- ❑ Beyond this presentation



pip install autogen-agentchat~=0.2

Key Concepts

Key concepts

Agents

- An entity that can send and receive **messages** to and from **other agents** in **its environment**
- **ConversableAgent** is a built-in agent that supports the following components (which can be turned on and off):
  - A list of LLMs
  - A code executor
  - A function/tool executor
  - A component for keeping human-in-the-loop
- LLMs, for example, enable agents to converse in natural languages and transform between structured and unstructured text
- The **generate_reply** method takes a question and generates a reply

ConversableAgent

```
import os

from autogen import ConversableAgent

agent = ConversableAgent(
    "chatbot",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ.get("OPENAI_API_KEY")}]},
    code_execution_config=False,  # Turn off code execution, by default it is off.
    function_map=None,  # No registered functions, by default it is None.
    human_input_mode="NEVER",  # Never ask for human input.
)
```

generate_reply

```
reply = agent.generate_reply(messages=[{"content": "Tell me a joke.", "role": "user"}])
print(reply)
```

```
Sure, here's a light-hearted joke for you:

Why don't scientists trust atoms?

Because they make up everything!
```

Key concepts

Personas and Conversations

- It is common for **agents** to embody a **persona**
- A **persona** is typically assigned to an agent using a **system prompt**
  - A system prompt is defined with the **system_message** during **ConversableAgent** instantiation
- Agents **participate in conversations** or chat with each other
- A **conversation** is a **sequence of messages** exchanged between agents
- Conversations are employed to **make progress on a task**
- A conversation is **started** using the **agent.initiate_chat(recipient, message, max_turns, …)** method
  - **recipient** is the agent receiving the message
  - **message** is the message being sent
  - **max_turns** indicates the number of conversation round trips

Let us have two agents put on a comedy show!

```
cathy = ConversableAgent(
    "cathy",
    system_message="Your name is Cathy and you are a part of a duo of comedians.",
    llm_config={"config_list": [{"model": "gpt-4", "temperature": 0.9, "api_key": os.environ.get("OPENAI_API_KEY")}]},
    human_input_mode="NEVER",  # Never ask for human input.
)

joe = ConversableAgent(
    "joe",
    system_message="Your name is Joe and you are a part of a duo of comedians.",
    llm_config={"config_list": [{"model": "gpt-4", "temperature": 0.7, "api_key": os.environ.get("OPENAI_API_KEY")}]},
    human_input_mode="NEVER",  # Never ask for human input.
)
```

```
result = joe.initiate_chat(cathy, message="Cathy, tell me a joke.", max_turns=2)
```

```
joe (to cathy):

Cathy, tell me a joke.

--------------------------------------------------------------------------------
cathy (to joe):

Sure, here's one for you:

Why don't scientists trust atoms?

Because they make up everything!

--------------------------------------------------------------------------------
joe (to cathy):

Haha, that's a good one, Cathy! Okay, my turn.

Why don't we ever tell secrets on a farm?

Because the potatoes have eyes, the corn has ears, and the beans stalk.

--------------------------------------------------------------------------------
cathy (to joe):

Haha, that's a great one! A farm is definitely not the place for secrets. Okay, my turn again.

Why couldn't the bicycle stand up by itself?

Because it was two-tired!

--------------------------------------------------------------------------------
```

First turn

Second turn

Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Terminating Conversations

Terminating Conversations (w/o human-in-the-loop)

- An any complex, autonomous workflows it's crucial to know **when/how to stop the workflow**
  - Task is completed
  - Process has consumed predetermined resources
- Two options
  - **As a conversation parameter** (parameter to **initiate_chat**)
    - **max_turns**: As we saw, this determines the number of conversation round trips
  - **As an agent parameter** (parameter to **ConversableAgent**)
    - **max_consecutive_auto_reply**: Triggers termination if the number of automatic responses to the **same sender** exceeds a threshold
    - **is_termination_msg**: Triggers termination if the received message satisfies a particular condition. More specifically, it is a function that **takes a message in the form of a dictionary and returns a boolean value** indicating if this received message is a termination message.
      - If **NONE** (**default**) is provided, **is_termination_msg** is internally set to **"TERMINATE"**
      - That is, by default, **this agent stops responding once it receives a "TERMINATE" message**

Using
max_consecutive_auto_reply

```python
joe = ConversableAgent(
    "joe",
    system_message="Your name is Joe and you are a part of a duo of comedians.",
    llm_config={"config_list": [{"model": "gpt-4", "temperature": 0.7, "api_key": os.environ.get("OPENAI_API_K
    human_input_mode="NEVER",  # Never ask for human input.
    max_consecutive_auto_reply=1,  # Limit the number of consecutive auto-replies.
)

result = joe.initiate_chat(cathy, message="Cathy, tell me a joke.")
```

```
joe (to cathy):

Cathy, tell me a joke.

--------------------------------------------------------------------------------
cathy (to joe):

Sure, here's one for you:

Why don't scientists trust atoms?

Because they make up everything!

--------------------------------------------------------------------------------
joe (to cathy):

Haha, that's a good one, Cathy! Okay, my turn.

Why don't we ever tell secrets on a farm?

Because the potatoes have eyes, the corn has ears, and the beans stalk.

--------------------------------------------------------------------------------
cathy (to joe):

Haha, that's a great one! A farm is definitely not the place for secrets. Okay, my turn again.

Why couldn't the bicycle stand up by itself?

Because it was two-tired!

--------------------------------------------------------------------------------
```

Note how Joe replies only one to Cathy

Using is_termination_msg

```
joe = ConversableAgent(
    "joe",
    system_message="Your name is Joe and you are a part of a duo of comedians.",
    llm_config={"config_list": [{"model": "gpt-4", "temperature": 0.7, "api_key": os.environ.get("OPENAI_API_KE
    human_input_mode="NEVER",  # Never ask for human input.
    is_termination_msg=lambda msg: "good bye" in msg["content"].lower(),
)

result = joe.initiate_chat(cathy, message="Cathy, tell me a joke and then say the words GOOD BYE.")
```

```
joe (to cathy):

Cathy, tell me a joke and then say the words GOOD BYE.

--------------------------------------------------------------------------------
cathy (to joe):

Why don't scientists trust atoms?

Because they make up everything!

GOOD BYE!

--------------------------------------------------------------------------------
```

Another example: Guess the number game

```
import os

from autogen import ConversableAgent

agent_with_number = ConversableAgent(
    "agent_with_number",
    system_message="You are playing a game of guess-my-number. You have the "
    "number 53 in your mind, and I will try to guess it. "
    "If I guess too high, say 'too high', if I guess too low, say 'too low'. ",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
    is_termination_msg=lambda msg: "53" in msg["content"],  # terminate if the number is guessed by the other a
    human_input_mode="NEVER",  # never ask for human input
)

agent_guess_number = ConversableAgent(
    "agent_guess_number",
    system_message="I have a number in my mind, and you will try to guess it. "
    "If I say 'too high', you should guess a lower number. If I say 'too low', "
    "you should guess a higher number. ",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
    human_input_mode="NEVER",
)

result = agent_with_number.initiate_chat(
    agent_guess_number,
    message="I have a number between 1 and 100. Guess it!",
)
```

```
agent_with_number (to agent_guess_number):

I have a number between 1 and 100. Guess it!

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 50?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too low.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 75?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too high.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 63?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too high.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 57?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too high.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 54?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too high.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 52?

--------------------------------------------------------------------------------
agent_with_number (to agent_guess_number):

Too low.

--------------------------------------------------------------------------------
agent_guess_number (to agent_with_number):

Is it 53?

--------------------------------------------------------------------------------
```

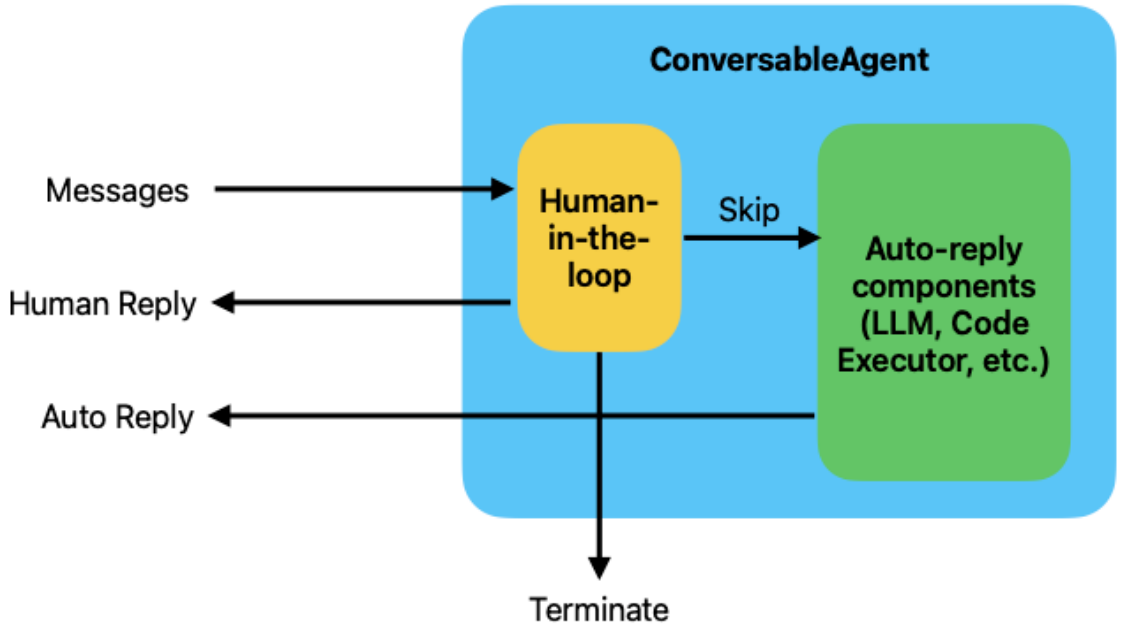
Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Human-in-the-loop

Human-in-the-loop

Human-in-the-loop

- **Many applications require human feedback** to steer agents in the right direction, specify goals, or terminate conversations.
- AutoGen offers this capability via Human input modes



Human input modes

- **NEVER**: human input is **never** requested. Meant for fully autonomous agents.
- **ALWAYS**: human input is **always** requested (**max_consecutive_auto_reply** is ignored) and the human can choose to either
  - **Do nothing** and trigger an auto-reply (i.e., the agent replies)
  - **Reply** to the message
  - **Terminate** the conversation (by typing **exit**)
- **TERMINATE** (**default**): **human input is only requested when a termination condition is met**. If the human chooses to reply, the conversation continues and the counter used by **max_consecutive_auto_reply** is reset.

human_input_mode = ALWAYS

- No LLM used for **human_proxy**, so this is a human <-> agent conversation

```
human_proxy = ConversableAgent(
    "human_proxy",
    llm_config=False,  # no LLM used for human proxy
    human_input_mode="ALWAYS",  # always ask for human input
)

# Start a chat with the agent with number with an initial guess.
result = human_proxy.initiate_chat(
    agent_with_number,  # this is the same agent with the number as before
    message="10",
)
```

Human is prompted to enter a response each time

```
human_proxy (to agent_with_number):

10

--------------------------------------------------------------------------------
agent_with_number (to human_proxy):

Too low.

--------------------------------------------------------------------------------
human_proxy (to agent_with_number):

79

--------------------------------------------------------------------------------
agent_with_number (to human_proxy):

Too high.

--------------------------------------------------------------------------------
human_proxy (to agent_with_number):

76

--------------------------------------------------------------------------------
agent_with_number (to human_proxy):

Too high.

--------------------------------------------------------------------------------
human_proxy (to agent_with_number):

I give up

--------------------------------------------------------------------------------
agent_with_number (to human_proxy):

That's okay! The number I was thinking of was 53.

--------------------------------------------------------------------------------
```

Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

human_input_mode = TERMINATE

- **Human input is requested when a termination condition is triggered**, so need to pay attention to those conditions.
- **If the human chooses to reply,** the agent's reply (auto reply) **counter is reset**
- **If the human chooses to skip**, the **agent replies** and the agent's reply (auto reply) **counter is incremented**.
- If the human chooses to terminate, the conversation ends

```
agent_with_number = ConversableAgent(
    "agent_with_number",
    system_message="You are playing a game of guess-my-number. "
    "In the first game, you have the "
    "number 53 in your mind, and I will try to guess it. "
    "If I guess too high, say 'too high', if I guess too low, say 'too low'. ",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
    max_consecutive_auto_reply=1,  # maximum number of consecutive auto-replies before asking for human input
    is_termination_msg=lambda msg: "53" in msg["content"],  # terminate if the number is guessed by the other
    human_input_mode="TERMINATE",  # ask for human input until the game is terminated
)

agent_guess_number = ConversableAgent(
    "agent_guess_number",
    system_message="I have a number in my mind, and you will try to guess it. "
    "If I say 'too high', you should guess a lower number. If I say 'too low', "
    "you should guess a higher number. ",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
    human_input_mode="NEVER",
)

result = agent_with_number.initiate_chat(
    agent_guess_number,
    message="I have a number between 1 and 100. Guess it!",
)
```

agent_with_number (to agent_guess_number):

I have a number between 1 and 100. Guess it!

---

agent_guess_number (to agent_with_number):

Is it 50?

>>>>>>>> USING AUTO REPLY...
agent_with_number (to agent_guess_number):

Too low.

---

agent_guess_number (to agent_with_number):

Is it 75?

---

agent_with_number (to agent_guess_number):

It is too high my friend.

Human answers

---

agent_guess_number (to agent_with_number):

Is it 60?

>>>>>>>> USING AUTO REPLY...
agent_with_number (to agent_guess_number):

Too high.

---

agent_guess_number (to agent_with_number):

Is it 55?

---

agent_with_number (to agent_guess_number):

still too high, but you are very close.

Human answers

---

agent_guess_number (to agent_with_number):

Is it 52?

>>>>>>>> USING AUTO REPLY...
agent_with_number (to agent_guess_number):

Too low.

---

agent_guess_number (to agent_with_number):

Is it 54?

---

agent_with_number (to agent_guess_number):

Almost there!

Human answers

---

agent_guess_number (to agent_with_number):

Is it 53?

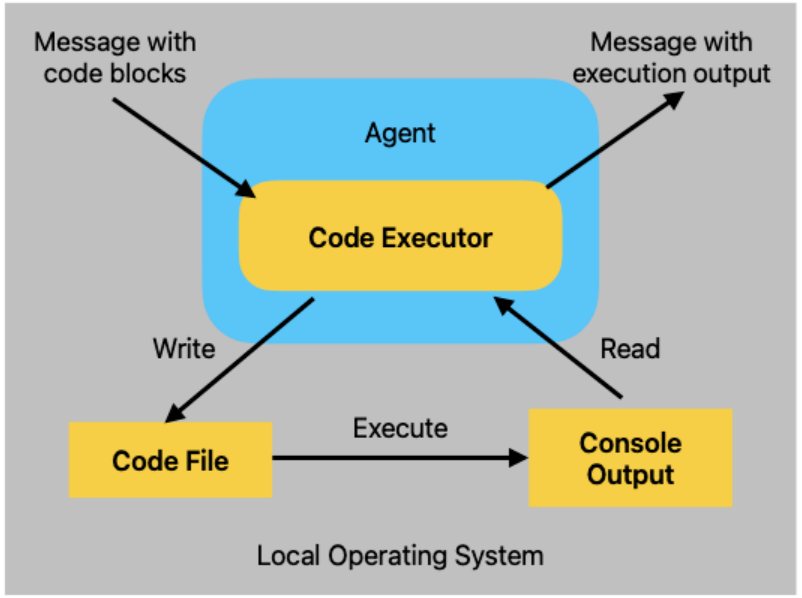
Oliva, Alware Leadership Bootcamp, Toronto, Canada, 2024

Code Executors

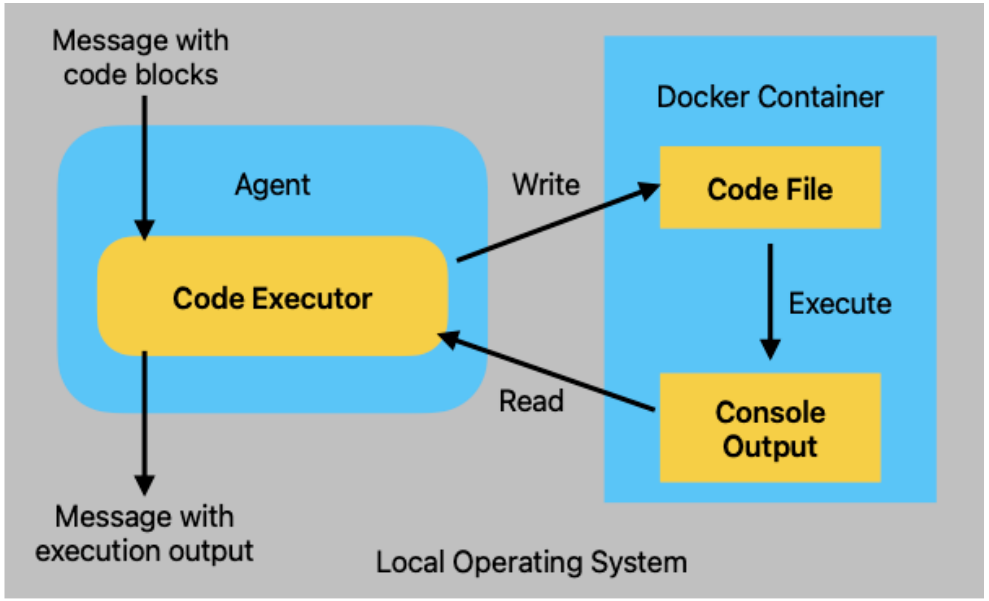
Code Executors

- A **code executor** is a component that takes **input messages containing code blocks**, performs **code execution**, and **outputs messages** with the results
- Two types of code executors
  - **Command Line**: runs on a **shell**, each **code block is executed in a new process** (stateless)
  - **Jupyter Kernel**: runs on a **stateful jupyter kernel** (e.g., you can define one variable in a code block and use it in another block)
- Each code executor can **run either locally or on a Docker** container

Local setup



Docker setup



Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Command Line Executor with a Local Setup

```python
import tempfile

from autogen import ConversableAgent
from autogen.coding import LocalCommandLineCodeExecutor

# Create a temporary directory to store the code files.
temp_dir = tempfile.TemporaryDirectory()

# Create a local command line code executor.
executor = LocalCommandLineCodeExecutor(
    timeout=10,  # Timeout for each code execution in seconds.
    work_dir=temp_dir.name,  # Use the temporary directory to store the code files.
)

# Create an agent with code executor configuration.
code_executor_agent = ConversableAgent(
    "code_executor_agent",
    llm_config=False,  # Turn off LLM for this agent.
    code_execution_config={"executor": executor},  # Use the local command line code executor.
    human_input_mode="ALWAYS",  # Always take human input for this agent for safety.
)
```

```python
message_with_code_block = """This is a message with code block.
The code block is below:
```python
import numpy as np
import matplotlib.pyplot as plt
x = np.random.randint(0, 100, 100)
y = np.random.randint(0, 100, 100)
plt.scatter(x, y)
plt.savefig('scatter.png')
print('Scatter plot saved to scatter.png')
```
This is the end of the message.
"""

# Generate a reply for the given code.
reply = code_executor_agent.generate_reply(messages=[{"role": "user", "content": message_with_code_bl
print(reply)
```

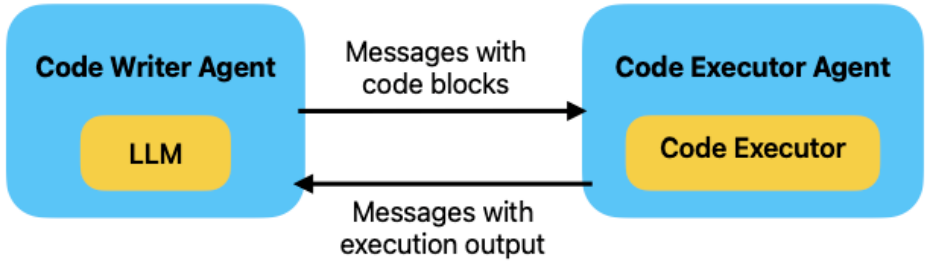
```
>>>>>>>> NO HUMAN INPUT RECEIVED.

>>>>>>>> USING AUTO REPLY...

>>>>>>>> EXECUTING CODE BLOCK (inferred language is python)...
exitcode: 0 (execution succeeded)
Code output:
Scatter plot saved to scatter.png
```

Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

A more interesting example…




```python
import tempfile

from autogen import ConversableAgent
from autogen.coding import LocalCommandLineCodeExecutor

# Create a temporary directory to store the code files.
temp_dir = tempfile.TemporaryDirectory()

# Create a local command line code executor.
executor = LocalCommandLineCodeExecutor(
    timeout=10,  # Timeout for each code execution in seconds.
    work_dir=temp_dir.name,  # Use the temporary directory to store the code files.
)

# Create an agent with code executor configuration.
code_executor_agent = ConversableAgent(
    "code_executor_agent",
    llm_config=False,  # Turn off LLM for this agent.
    code_execution_config={"executor": executor},  # Use the local command line code executor.
    human_input_mode="ALWAYS",  # Always take human input for this agent for safety.
)
```

Same as before

```python
# The code writer agent's system message is to instruct the LLM on how to use
# the code executor in the code executor agent.
code_writer_system_message = """You are a helpful AI assistant.
Solve tasks using your coding and language skills.
In the following cases, suggest python code (in a python coding block) or shell script (in a sh codin
1. When you need to collect info, use the code to output the info you need, for example, browse or se
2. When you need to perform some task with code, use the code to perform the task and output the resu
Solve the task step by step if you need to. If a plan is not provided, explain your plan first. Be cl
When using code, you must indicate the script type in the code block. The user cannot provide any oth
If you want the user to save the code in a file before executing it, put # filename: <filename> insid
If the result indicates there is an error, fix the error and output the code again. Suggest the full
When you find an answer, verify the answer carefully. Include verifiable evidence in your response if
Reply 'TERMINATE' in the end when everything is done.
"""

code_writer_agent = ConversableAgent(
    "code_writer_agent",
    system_message=code_writer_system_message,
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
    code_execution_config=False,  # Turn off code execution for this agent.
)
```

```python
chat_result = code_executor_agent.initiate_chat(
    code_writer_agent,
    message="Write Python code to calculate the 14th Fibonacci number.",
)
```

code_executor_agent (*to* code_writer_agent):

Write Python code to calculate the 14th Fibonacci number.

>>>>>>>> USING AUTO REPLY...
code_writer_agent (*to* code_executor_agent):

Sure, here is a Python code snippet to calculate the 14th Fibonacci number. The Fibonacci series is a

```python
def fibonacci(n):
    if(n <= 0):
        return "Input should be a positive integer."
    elif(n == 1):
        return 0
    elif(n == 2):
        return 1
    else:
        fib = [0, 1]
        for i in range(2, n):
            fib.append(fib[i-1] + fib[i-2])
        return fib[n-1]

print(fibonacci(14))
```

This Python code defines a function `fibonacci(n)` which computes the n-th Fibonacci number. The funct

---

>>>>>>>> NO HUMAN INPUT RECEIVED.

>>>>>>>> USING AUTO REPLY...

>>>>>>>> EXECUTING CODE BLOCK (inferred language is python)...
code_executor_agent (*to* code_writer_agent):

exitcode: 0 (execution succeeded)
Code output:
233

>>>>>>>> USING AUTO REPLY...
code_writer_agent (*to* code_executor_agent):

Great, the execution was successful and the 14th Fibonacci number is 233. The sequence goes as follows

I hope this meets your expectations. If you have any other concerns or need further computations, feel

TERMINATE

```
code_executor_agent (to code_writer_agent):

Write Python code to calculate the 14th Fibonacci number.

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

>>>>>>>> USING AUTO REPLY...
code_writer_agent (to code_executor_agent):

Sure, here is a Python code snippet to calculate the 14th Fibonacci number. The Fibonacci series is a

```python
def fibonacci(n):
    if(n <= 0):
        return "Input should be a positive integer."
    elif(n == 1):
        return 0
    elif(n == 2):
        return 1
    else:
        fib = [0, 1]
        for i in range(2, n):
            fib.append(fib[i-1] + fib[i-2])
        return fib[n-1]

print(fibonacci(14))
```

This Python code defines a function `fibonacci(n)` which computes the n-th Fibonacci number. The funct
```

```
--------------------------------------------------------------------------------

>>>>>>>> NO HUMAN INPUT RECEIVED.

>>>>>>>> USING AUTO REPLY...

>>>>>>>> EXECUTING CODE BLOCK (inferred language is python)...
code_executor_agent (to code_writer_agent):

exitcode: 0 (execution succeeded)
Code output:
233


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

>>>>>>>> USING AUTO REPLY...
code_writer_agent (to code_executor_agent):

Great, the execution was successful and the 14th Fibonacci number is 233. The sequence goes as follows

I hope this meets your expectations. If you have any other concerns or need further computations, feel

TERMINATE

--------------------------------------------------------------------------------
```

Is the conversation over?

It actually takes a few more turns for the conversation to end…

```
--------------------------------------------------------------------------------
Replying as code_executor_agent. Provide feedback to code_writer_agent. Press enter to skip and use auto-reply, or type 'exit' to end the conversation:

>>>>>>>> NO HUMAN INPUT RECEIVED.

>>>>>>>> USING AUTO REPLY...
code_executor_agent (to code_writer_agent):



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

>>>>>>>> USING AUTO REPLY...
code_writer_agent (to code_executor_agent):

TERMINATE

--------------------------------------------------------------------------------
Replying as code_executor_agent. Provide feedback to code_writer_agent. Press enter to skip and use auto-reply, or type 'exit' to end the conversation:

>>>>>>>> NO HUMAN INPUT RECEIVED.
```

code_executor_agent should **define an explicit termination function** (e.g., last word in received message is "TERMINATE")

Command Line Executor with a Docker Setup

```python
from autogen.coding import DockerCommandLineCodeExecutor

# Create a temporary directory to store the code files.
temp_dir = tempfile.TemporaryDirectory()

# Create a Docker command line code executor.
executor = DockerCommandLineCodeExecutor(
    image="python:3.12-slim",  # Execute code using the given docker image name.
    timeout=10,  # Timeout for each code execution in seconds.
    work_dir=temp_dir.name,  # Use the temporary directory to store the code files.
)

# Create an agent with code executor configuration that uses docker.
code_executor_agent_using_docker = ConversableAgent(
    "code_executor_agent_docker",
    llm_config=False,  # Turn off LLM for this agent.
    code_execution_config={"executor": executor},  # Use the docker command line code executor.
    human_input_mode="ALWAYS",  # Always take human input for this agent for safety.
)

# When the code executor is no longer used, stop it to release the resources.
# executor.stop()
```

Tool Use

Tool Use

What are tools?

- **Tools are pre-defined functions that agents can use**
  - Searching the web, performing calculations, reading files, or calling remote APIs
- Tools provide more control over the agent's actions (including code generation)
- Tool use is currently **only available for LLMs that support OpenAI-compatible tool call API**.

How to create tools?

- Tools can be created as regular Python functions
- Make sure to **use type hints for arguments and return value of functions**
- Also supports pydantic (for more complex schema definitions)

Registering tools

- **A tool must be registered with two agents for it to be useful in a conversation**.
- The agent registered with the tool's signature through **register_for_llm** can create a tool call
- The agent registered with the tool's function object through **register_for_execution** can execute the call.
- Tool usage and code execution can be "hidden" within a single agent via nested chats

Let’s create an agent that can call a calculator
First let’s define the python function…

```python
from typing import Annotated, Literal

Operator = Literal["+", "-", "*", "/"]


def calculator(a: int, b: int, operator: Annotated[Operator, "operator"]) -> int:
    if operator == "+":
        return a + b
    elif operator == "-":
        return a - b
    elif operator == "*":
        return a * b
    elif operator == "/":
        return int(a / b)
    else:
        raise ValueError("Invalid operator")
```

Registering the tool…
(no change in how an agent is instantiated)

```python
import os

from autogen import ConversableAgent

# Let's first define the assistant agent that suggests tool calls.
assistant = ConversableAgent(
    name="Assistant",
    system_message="You are a helpful AI assistant. "
    "You can help with simple calculations. "
    "Return 'TERMINATE' when the task is done.",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_API_KEY"]}]},
)

# The user proxy agent is used for interacting with the assistant agent
# and executes tool calls.
user_proxy = ConversableAgent(
    name="User",
    llm_config=False,
    is_termination_msg=lambda msg: msg.get("content") is not None and "TERMINATE" in msg["content"],
    human_input_mode="NEVER",
)

# Register the tool signature with the assistant agent.
assistant.register_for_llm(name="calculator", description="A simple calculator")(calculator)

# Register the tool function with the user proxy agent.
user_proxy.register_for_execution(name="calculator")(calculator)
```

The tool's schema is auto-generated by AutoGen from the function's typehints…

```
assistant.llm_config["tools"]
```

```
[{'type': 'function',
  'function': {'description': 'A simple calculator',
   'name': 'calculator',
   'parameters': {'type': 'object',
    'properties': {'a': {'type': 'integer', 'description': 'a'},
     'b': {'type': 'integer', 'description': 'b'},
     'operator': {'enum': ['+', '-', '*', '/'],
      'type': 'string',
      'description': 'operator'}},
    'required': ['a', 'b', 'operator']}}}]
```

```
chat_result = user_proxy.initiate_chat(assistant, message="What is (44232 + 13312 / (232 - 32)) * 5?")
```

```
User (to Assistant):

What is (44232 + 13312 / (232 - 32)) * 5?

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

>>>>>>>> USING AUTO REPLY...
Assistant (to User):

***** Suggested tool call (call_4rElPoLggOYJmkUutbGaSTX1): calculator *****
Arguments:
{
  "a": 232,
  "b": 32,
  "operator": "-"
}
**********************************************************************

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

>>>>>>>> EXECUTING FUNCTION calculator...
User (to Assistant):

User (to Assistant):

***** Response from calling tool (call_4rElPoLggOYJmkUutbGaSTX1) *****
200
**********************************************************************

--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------

>>>>>>>> USING AUTO REPLY...
Assistant (to User):

***** Suggested tool call (call_SGtr8tK9A4iOCJGdCqkKR2Ov): calculator *****
Arguments:
{
  "a": 13312,
  "b": 200,
  "operator": "/"
}
**********************************************************************

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

>>>>>>>> EXECUTING FUNCTION calculator...
User (to Assistant):

User (to Assistant):

***** Response from calling tool (call_SGtr8tK9A4iOCJGdCqkKR2Ov) *****
66
**********************************************************************

--------------------------------------------------------------------------------
```

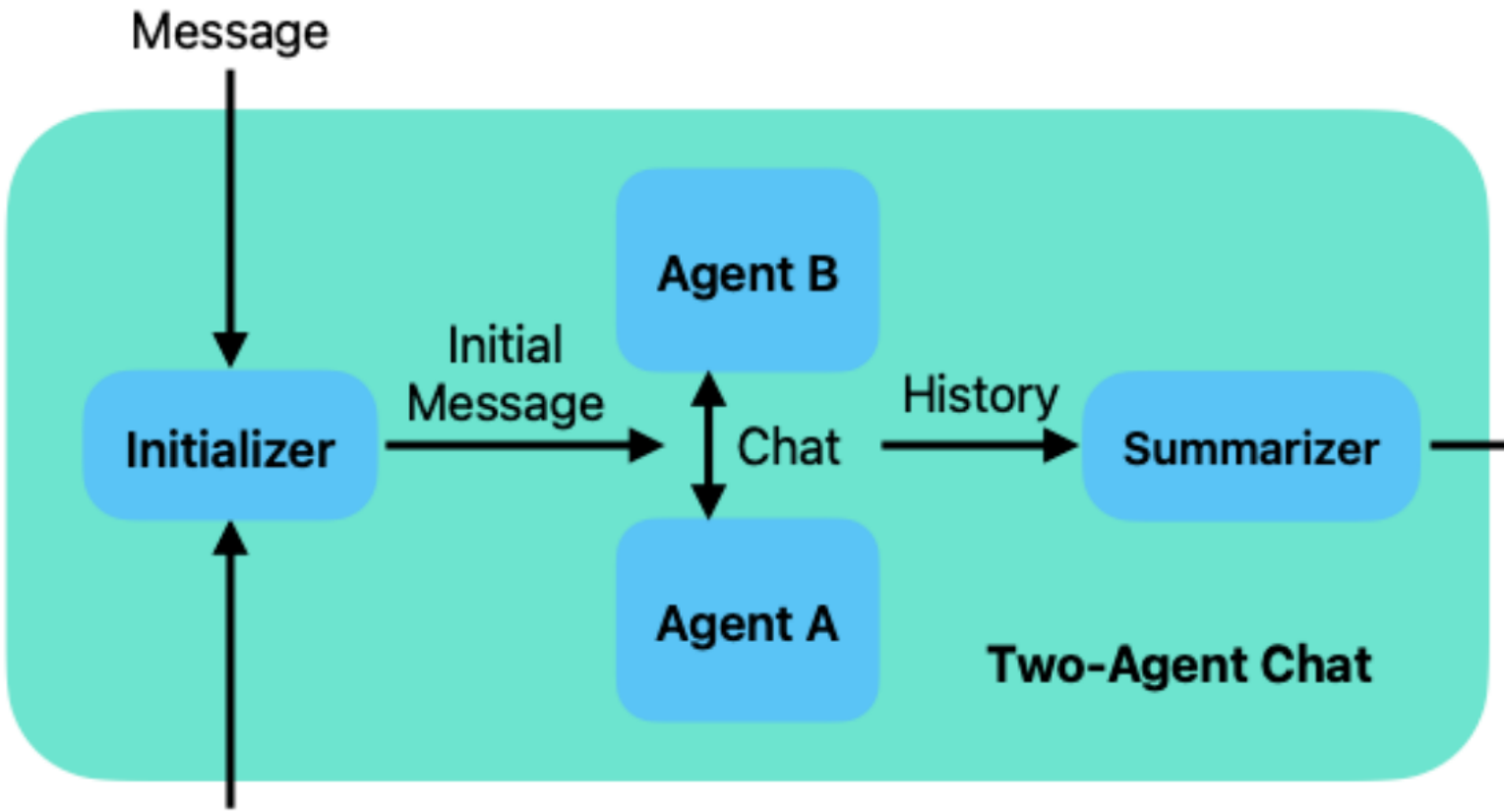
Conversation Patterns

Overview

- **Two-agent chat**: the simplest form of conversation pattern where two agents chat with each other.
- **Sequential chat**: a sequence of chats between two agents, chained together by a carryover mechanism, which brings the summary of the previous chat to the context of the next chat.
- **Group Chat**: a single chat involving more than two agents.
  - Several strategies can be used to define the next speaker (agent): round_robin, random, manual (human selection), and auto (Default, using an LLM to decide).
  - Selection of the next speaker can be constrained using **allowed** and **disallowed** speaker transitions
  - Selection of the next speaker can be done with a user-defined function (e.g., allowing a deterministic workflow among agents)
- **Nested Chat**: package a workflow into a single agent for reuse in a larger workflows.

Two-Agent Chat

E.g., "What is triangle inequality?"



Contains:

- Conversation history
- Human input
- Token cost
- ….

E.g., summary_method: reflection_with_llm

- Takes a list of messages from the conversation and summarizes them using a call to an LLM (recipient's LLM).
- Prompt: "Summarize the takeaway from the conversation. Do not add any introductory phrases." (can be customized)

```python
import os

from autogen import ConversableAgent

student_agent = ConversableAgent(
    name="Student_Agent",
    system_message="You are a student willing to learn.",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_AP
)
teacher_agent = ConversableAgent(
    name="Teacher_Agent",
    system_message="You are a math teacher.",
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_AP
)

chat_result = student_agent.initiate_chat(
    teacher_agent,
    message="What is triangle inequality?",
    summary_method="reflection_with_llm",
    max_turns=2,
)
```

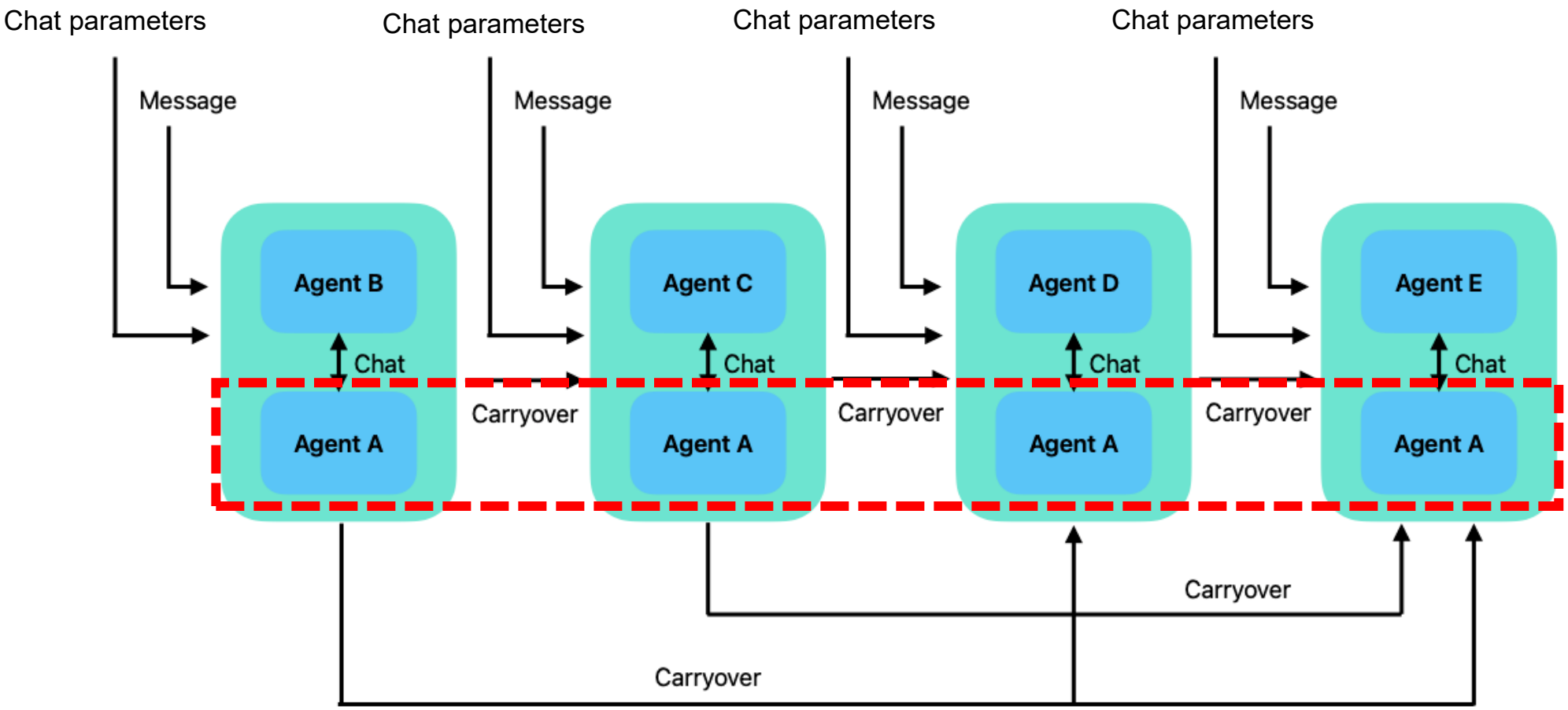
```python
print(chat_result.summary)
```

```
The triangle inequality theorem states that in a triangle, the sum of the lengths of
```

Sequential Chats

This pattern is useful for complex task that can be broken down into interdependent sub-tasks

- **Carryover (conversation summary) accumulates as the conversation moves forward**, so each **subsequent chat starts with all the carryovers from previous chats**.



Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Example: **arithmetic operations with agents**

 # The Number Agent always returns the same numbers.

 # The Adder Agent adds 1 to each number it receives.

 # The Multiplier Agent multiplies each number it receives by 2.

 # The Subtracter Agent subtracts 1 from each number it receives.

 # The Divider Agent divides each number it receives by 2.

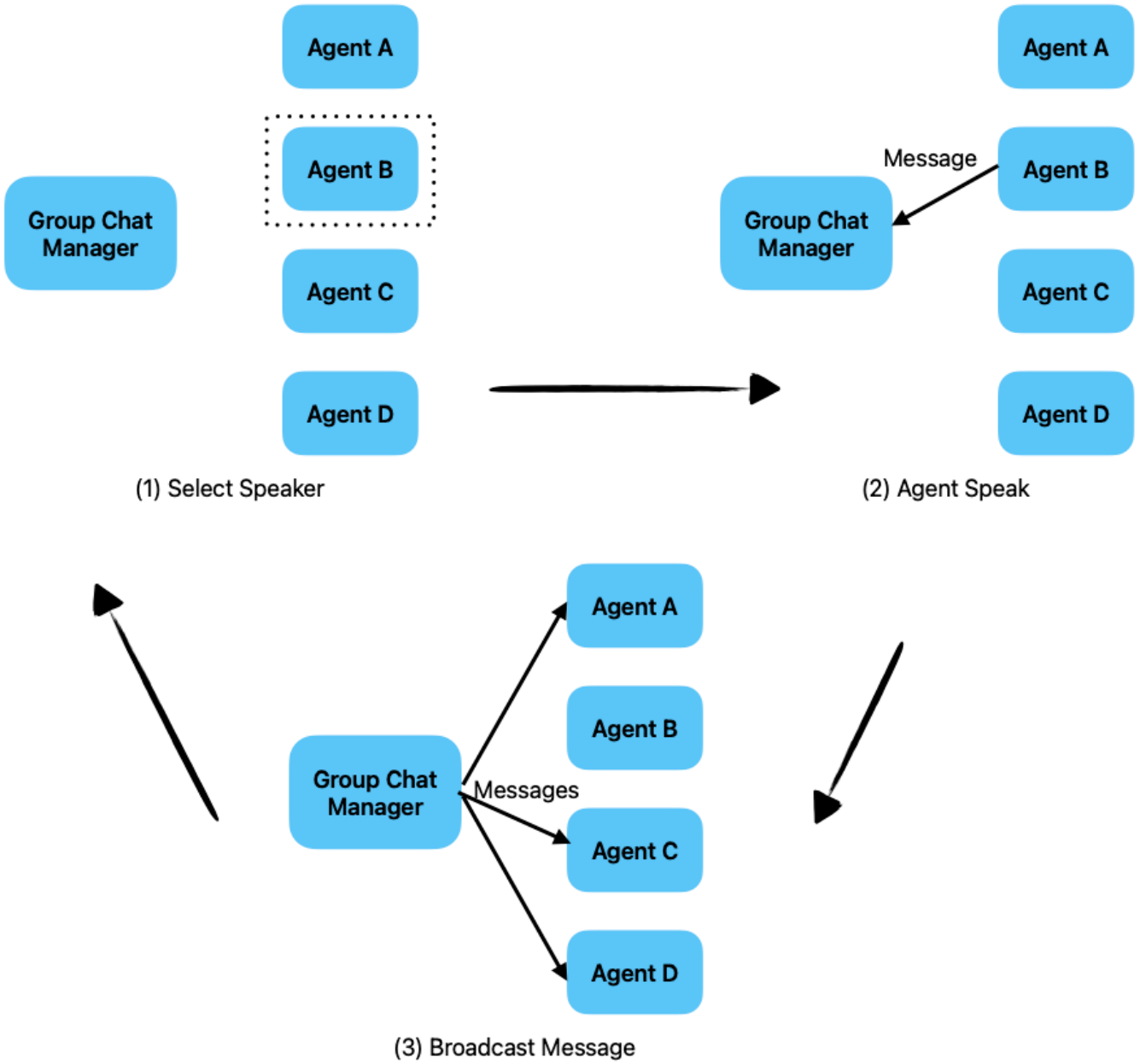
```
# Start a sequence of two-agent chats.
# Each element in the list is a dictionary that specifies the arguments
# for the initiate_chat method.
chat_results = number_agent.initiate_chats(
    [
        {
            "recipient": adder_agent,
            "message": "14",
            "max_turns": 2,
            "summary_method": "last_msg",
        },
        {
            "recipient": multiplier_agent,
            "message": "These are my numbers",
            "max_turns": 2,
            "summary_method": "last_msg",
        },
        {
            "recipient": subtracter_agent,
            "message": "These are my numbers",
            "max_turns": 2,
            "summary_method": "last_msg",
        },
        {
            "recipient": divider_agent,
            "message": "These are my numbers",
            "max_turns": 2,
            "summary_method": "last_msg",
        },
    ]
)
```

```
print("First Chat Summary: ", chat_results[0].summary)
print("Second Chat Summary: ", chat_results[1].summary)
print("Third Chat Summary: ", chat_results[2].summary)
print("Fourth Chat Summary: ", chat_results[3].summary)
```

```
First Chat Summary:  16
Second Chat Summary:  64
Third Chat Summary:  14
62
Fourth Chat Summary:  4
16
3.5
15.5
```

Group Chats

- All agents contribute to a single conversation thread (chat)
  - Agents share the same context
- A **GroupChatManager** **decides who will speak next** using one of these strategies:
  - **round_robin**
  - **random**
  - **manual** (human selection)
  - **auto** (default, LLM decides).
- The selection of the next speaker can be constrained
- The selection of the next speaker can be customized with a Python function



Basic recipe for group chat (using ‘auto’)

1) Describe the Agents

- To help the **GroupChatManager** select the next agent, we **add a description to the agents** that will engage in the group chat.
- If a description is not provided, the **GroupChatManager** will use the agents’ **system_message** (system prompt) to decide the order, which might not be the best choice.

```
# The `description` attribute is a string that describes the agent.
# It can also be set in `ConversableAgent` constructor.
adder_agent.description = "Add 1 to each input number."
multiplier_agent.description = "Multiply each input number by 2."
subtracter_agent.description = "Subtract 1 from each input number."
divider_agent.description = "Divide each input number by 2."
number_agent.description = "Return the numbers given."
```

Basic recipe for group chat (using ‘auto’)

2) Instantiate a GroupChat

- Defines the basic parameters of the chat
- The **agents** list defines the list of agents who will chat
  - If **round_robin** is used, the list order is respected
- The **speaker_selection_method** determines the method for selecting the next speaker (**omitted below**, **defaults to auto**)
- The **messages** list acts as the **starting history or context for the conversation among the agents**.
  - It helps establish any predefined interactions, setup information, or introductory dialogue that the agents can reference during the chat. (empty in this example)
- The **max_rounds** parameter defines the number of chat rounds (“Select speaker -> agent speaks -> message is broadcasted”).

```
from autogen import GroupChat

group_chat = GroupChat(
    agents=[adder_agent, multiplier_agent, subtracter_agent, divider_agent, number_
    messages=[],
    max_round=6,
)
```

Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Basic recipe for group chat (using ‘auto’)

3) Instantiate a GroupChatManager

- A **GroupChatManager** takes a **GroupChat** as input (i.e., the group chat that it will manage)
- The **auto** mode uses an LLM to select the next speaker based on their descriptions, so we need to **specify an LLM for this agent**

```
from autogen import GroupChatManager

group_chat_manager = GroupChatManager(
    groupchat=group_chat,
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_AP
)
```

Basic recipe for group chat (using ‘auto’)

4) Initiate the chat

- We initiate the chat as usual in a two-agent style
- In this example, one of the agents in the group (number agent) sends a message to the group chat manager
- The group chat manager will then run the group chat internally and terminate the two-agent chat when the internal group chat is done.
- Since the **number_agent** is selected to speak by us, it counts as the first round of the group chat.

```
chat_result = number_agent.initiate_chat(
    group_chat_manager,
    message="My number is 3, I want to turn it into 13.",
    summary_method="reflection_with_llm",
)
```

In practice, it is as if a team member is asking a question the whole team

(and a moderator coordinates the conversation)!

```
Number_Agent (to chat_manager):

My number is 3, I want to turn it into 13.

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

6

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

7

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

14

--------------------------------------------------------------------------------
Subtracter_Agent (to chat_manager):

13

--------------------------------------------------------------------------------
Number_Agent (to chat_manager):

13

--------------------------------------------------------------------------------
```

Oliva, AIware Leadership Bootcamp, Toronto, Canada, 2024

Tailoring Group Chats: Sending Introductions

- The description field of agents helps the **GroupChatManager** select the next agent
  - Does not help the participating agents to know about each other
- If **send_introductions** is set to **True**, the agents will introduce themselves to other agents in the same chat
  - Under the hood, the **GroupChatManager** sends a message containing the agents' names and descriptions to all agents in the group chat before the group chat starts.

```
group_chat_with_introductions = GroupChat(
    agents=[adder_agent, multiplier_agent, subtracter_agent, divider_agent, number_
    messages=[],
    max_round=6,
    send_introductions=True,
)
```

Tailoring Group Chats: Constraining Speaker Selection

- Group chat is a powerful conversation pattern, but it can be **hard to control** if the number of participating agents is large.
- AutoGen provides a way to constrain the selection of the next speaker by using the **allowed_or_disallowed_speaker_transitions** and **speaker_transition_type** argument of the **GroupChat** class.
- Let us see an example…

```
allowed_transitions = {
    number_agent: [adder_agent, number_agent],
    adder_agent: [multiplier_agent, number_agent],
    subtracter_agent: [divider_agent, number_agent],
    multiplier_agent: [subtracter_agent, number_agent],
    divider_agent: [adder_agent, number_agent],
}
```

```
constrained_graph_chat = GroupChat(
    agents=[adder_agent, multiplier_agent, subtracter_agent, divider_agent, number_
    allowed_or_disallowed_speaker_transitions=allowed_transitions,
    speaker_transitions_type="allowed",
    messages=[],
    max_round=12,
    send_introductions=True,
)

constrained_group_chat_manager = GroupChatManager(
    groupchat=constrained_graph_chat,
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_AP
)

chat_result = number_agent.initiate_chat(
    constrained_group_chat_manager,
    message="My number is 3, I want to turn it into 10. Once I get to 10, keep it t
    summary_method="reflection_with_llm",
)
```

```
Number_Agent (to chat_manager):

My number is 3, I want to turn it into 10. Once I get to 10, keep it there.

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

4

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

8

--------------------------------------------------------------------------------
Subtracter_Agent (to chat_manager):

7

--------------------------------------------------------------------------------
Divider_Agent (to chat_manager):

3.5

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

4.5

--------------------------------------------------------------------------------
```

```
Multiplier_Agent (to chat_manager):

9

--------------------------------------------------------------------------------
Subtracter_Agent (to chat_manager):

8

--------------------------------------------------------------------------------
Divider_Agent (to chat_manager):

4

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

5

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

10

--------------------------------------------------------------------------------
Number_Agent (to chat_manager):

10

--------------------------------------------------------------------------------
```


Group Chats as part of Sequential Chats

```
# Let's use the group chat with introduction messages created above.
group_chat_manager_with_intros = GroupChatManager(
    groupchat=group_chat_with_introductions,
    llm_config={"config_list": [{"model": "gpt-4", "api_key": os.environ["OPENAI_AP
)

# Start a sequence of two-agent chats between the number agent and
# the group chat manager.
chat_result = number_agent.initiate_chats(
    [
        {
            "recipient": group_chat_manager_with_intros,
            "message": "My number is 3, I want to turn it into 13.",
        },
        {
            "recipient": group_chat_manager_with_intros,
            "message": "Turn this number to 32.",
        },
    ]
)
```

```
********************************************************************************
Start a new chat with the following message:
My number is 3, I want to turn it into 13.

With the following carryover:


********************************************************************************
Number_Agent (to chat_manager):

My number is 3, I want to turn it into 13.

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

6

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

7

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

14

--------------------------------------------------------------------------------
Subtracter_Agent (to chat_manager):

13
```

```
--------------------------------------------------------------------------------
Number_Agent (to chat_manager):

Your number is 13.

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

********************************************************************************
Start a new chat with the following message:
Turn this number to 32.

With the following carryover:
Your number is 13.

********************************************************************************
Number_Agent (to chat_manager):

Turn this number to 32.
Context:
Your number is 13.

--------------------------------------------------------------------------------
Multiplier_Agent (to chat_manager):

26

--------------------------------------------------------------------------------
Adder_Agent (to chat_manager):

14
```

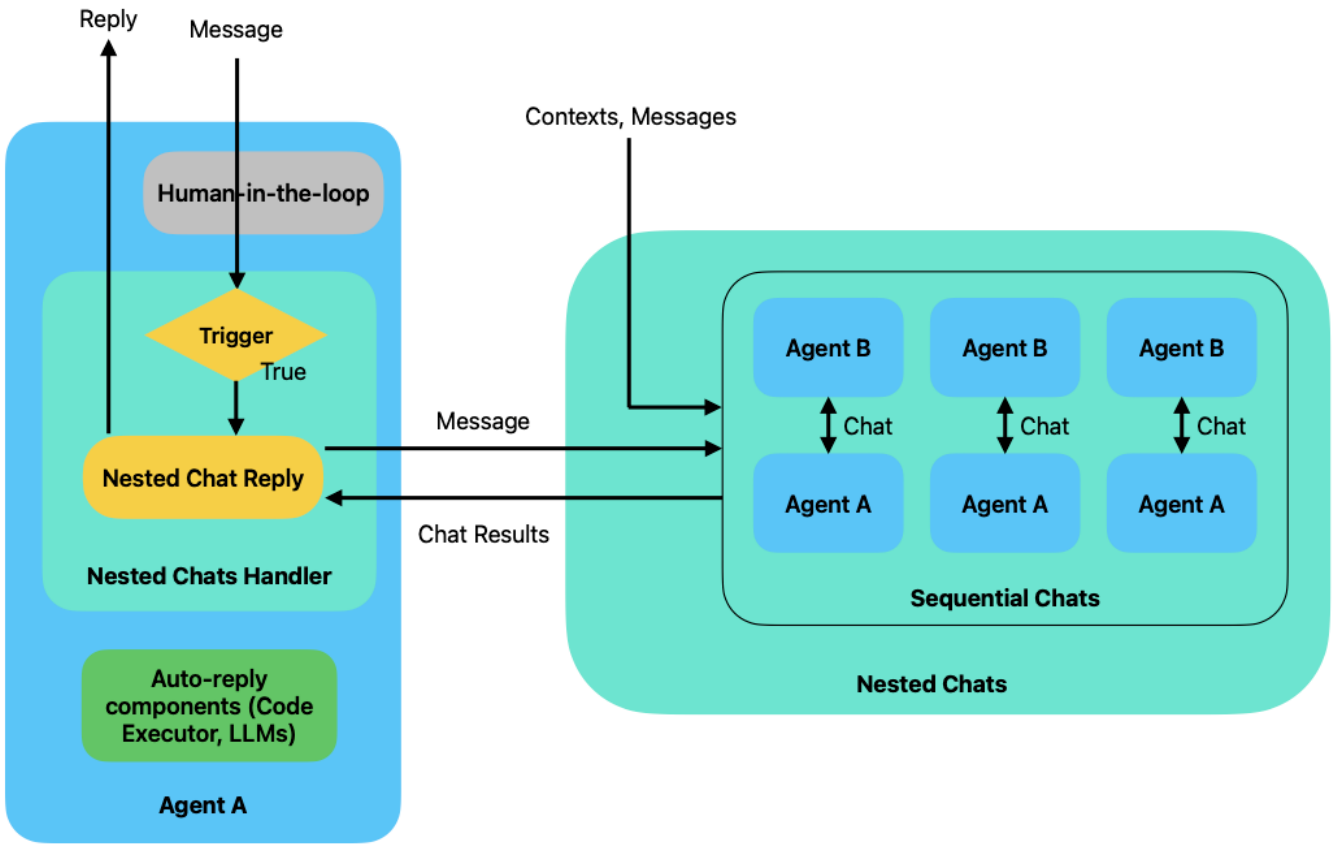
...

Nested Chats

- **Encapsulates a complex chat into an atomic unit** (think of it as a subworkflow node)
- Exposes a **single conversational interface**
  - Often needed for scenarios like question-answering bots and personal assistants

Mechanism:

- After passing the human-in-the-loop component, the nested chats handler checks if the message should trigger a nested chat based on conditions specified by the user.
- If the conditions are met, the nested chats handler starts a sequence of nested chats specified using the sequential chats pattern.
- In each of the nested chats, the sender agent is always the same agent that triggered the nested chats.
- In the end, the nested chat handler uses the results of the nested chats to produce a response to the original message.
- By default, the nested chat handler uses the summary of the last chat as the response.



Step 1. Define the Agents

```
human_proxy = ConversableAgent(
    name="HumanProxy",
    llm_config=False,
    human_input_mode="NEVER",
)

# Code Writer Agent: Generates Python code for the required task
code_writer_agent = ConversableAgent(
    name="CodeWriterAgent",
    system_message="You are a code writer. Generate Python code in Markdown format. Do not show the output or try
    to run the code",
    llm_config={"config_list": [gpt_4o_config]},
    human_input_mode="NEVER",
)

# Code Executor Agent: Executes the generated code locally in a sandboxed environment
code_executor_agent = ConversableAgent(
    name="CodeExecutorAgent",
    system_message="You are a code executor that runs Python code locally and reports the results.",
    code_execution_config={"use_docker": False, "work_dir": temp_dir},
    human_input_mode="ALWAYS"  # Keeps human verification for code execution
)
```

Step 2. Define the nested chats

Step 3. Register the nested chats

Step 4. Ask the question (mimicking a human)

```python
# Define the nested chat setup
nested_chats = [
    {
        "recipient": code_writer_agent,
        "summary_method": "last_msg",
        "max_turns": 1,
    },
    {
        "sender": code_writer_agent,
        "recipient": code_executor_agent,
        "message": "Execute the provided Python code and return the output.",
        "summary_method": "last_msg",
        "max_turns": 1,
    }
]

# Register the nested chats
human_proxy.register_nested_chats(
    nested_chats,
    trigger=lambda sender: sender not in [code_writer_agent, code_executor_agent]
)

# Example: Generating and executing code for a simple user query
reply = human_proxy.generate_reply(
    messages=[{"role": "user", "content": "Write code to turn the number 3 into 7 by adding 4."}]
)

print(f"The final answer is {reply}")
```

```
********************************************************************************
Starting a new chat....

********************************************************************************
HumanProxy (to CodeWriterAgent):

Write code to turn the number 3 into 7 by adding 4.

--------------------------------------------------------------------------------
CodeWriterAgent (to HumanProxy):

```python
# Start with the number 3
number = 3

# Add 4 to the number
number += 4

# The resulting number should be 7
print(number)
```

```

```
********************************************************************************
Starting a new chat....

********************************************************************************
CodeWriterAgent (to CodeExecutorAgent):

Execute the provided Python code and return the output.
Context:
```python
# Start with the number 3
number = 3

# Add 4 to the number
number += 4

# The resulting number should be 7
print(number)
```

Replying as CodeExecutorAgent. Provide feedback to CodeWriterAgent. Press enter

>>>>>>>> NO HUMAN INPUT RECEIVED.

>>>>>>>> USING AUTO REPLY...

>>>>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
CodeExecutorAgent (to CodeWriterAgent):

exitcode: 0 (execution succeeded)
Code output:
7
```

Agent Memory with Mem0

https://docs.mem0.ai/integrations/autogen#autogen

Mem0: How does it work?

- Mem0 leverages a **hybrid database approach** to manage and retrieve long-term memories for AI agents and assistants.
- **Each memory is associated with a unique identifier**, such as user_id/agent_id/session_id, allowing Mem0 to organize and access memories specific to an individual or context.

Adding memories

- When a message is added to the Mem0 using **add()** method, the **system extracts relevant facts and preferences and stores it across data stores**: a vector database, a key-value database, and a graph database.
- This hybrid approach ensures that different types of information are stored in the most efficient **manner**, making subsequent searches quick and effective.

Recalling memories

- When an AI agent or LLM needs to recall memories, it uses the **search()** method.
- **Mem0 then performs search across these data stores**, retrieving relevant information from each source.
- This **information is then passed through a scoring layer**, which evaluates their importance based on relevance, importance, and recency. This ensures that only the most personalized and useful context is surfaced.
- The retrieved memories can then be appended to the LLM's prompt as needed, making responses personalized and relevant.

Storing Conversations in Memory

Add conversation history to Mem0 for future reference:

```
conversation = [
    {"role": "assistant", "content": "Hi, I'm Best Buy's chatbot! How can I help you
    {"role": "user", "content": "I'm seeing horizontal lines on my TV."},
    {"role": "assistant", "content": "I'm sorry to hear that. Can you provide your T
    {"role": "user", "content": "It's a Sony - 77\" Class BRAVIA XR A80K OLED 4K UHD
    {"role": "assistant", "content": "Thank you for the information. Let's troublesh
]

memory_client.add(messages=conversation, user_id=USER_ID)
print("Conversation added to memory.")
```

Retrieving and Using Memory

Create a function to get context-aware responses based on user's question and previous interactions:

```
def get_context_aware_response(question):
    relevant_memories = memory_client.search(question, user_id=USER_ID)
    context = "\n".join([m["memory"] for m in relevant_memories])

    prompt = f"""Answer the user question considering the previous interactions:
    Previous interactions:
    {context}

    Question: {question}
    """

    reply = agent.generate_reply(messages=[{"content": prompt, "role": "user"}])
    return reply

# Example usage
question = "What was the issue with my TV?"
answer = get_context_aware_response(question)
print("Context-aware answer:", answer)
```

Multi-Agent Conversation

For more complex scenarios, you can create multiple agents:

```
manager = ConversableAgent(
    "manager",
    system_message="You are a manager who helps in resolving complex customer issues
    llm_config={"config_list": [{"model": "gpt-4", "api_key": OPENAI_API_KEY}]},
    human_input_mode="NEVER"
)

def escalate_to_manager(question):
    relevant_memories = memory_client.search(question, user_id=USER_ID)
    context = "\n".join([m["memory"] for m in relevant_memories])

    prompt = f"""
    Context from previous interactions:
    {context}

    Customer question: {question}

    As a manager, how would you address this issue?
    """

    manager_response = manager.generate_reply(messages=[{"content": prompt, "role":
    return manager_response

# Example usage
complex_question = "I'm not satisfied with the troubleshooting steps. What else can
manager_answer = escalate_to_manager(complex_question)
print("Manager's response:", manager_answer)
```

Overview of the session

- ❑ Introduction (5min)
- ❑ Deep dive into Microsoft AutoGen with examples
- ❑ **Creating an AutoGen playground for experimentation**
- ❑ Other agentic development platforms
- ❑ Standardization efforts for FM-powered Agents (5 min)
- ❑ Beyond this presentation (1 min)

Ollama

Ollama in a Nutshell

- **Ollama: Local AI Model Hub**
  - Ollama is a platform for discovering, running, and managing FMs (typically LLMs) directly on personal devices. It ensures privacy by operating offline and enables AI model use without internet connectivity.
- **Efficient Resource Utilization**
  - Ollama intelligently selects between CPU and GPU based on hardware availability, model size, and user configurations.
  - If a compatible GPU is available, Ollama defaults to it; otherwise, it uses the CPU.
  - For large models, it may split processing between GPU and CPU to optimize performance.
- **Accessible and Flexible**
  - With a user-friendly interface across operating systems, Ollama allows both developers and non-developers to experiment with powerful AI tools seamlessly on local machines.

Blog Discord GitHub

Search models

Models Sign in Download

Get up and running with large language models.

Run Llama 3.2, Phi 3, Mistral, Gemma 2, and other models. Customize and create your own.

Download ↓

Available for macOS, Linux, and Windows

Download Ollama

macOS

Linux

Windows

Install with one command:

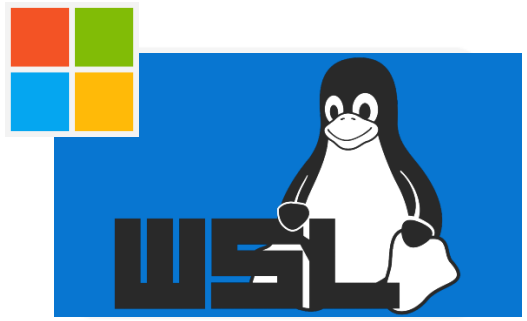
```
curl -fsSL https://ollama.com/install.sh | sh
```

View script source • Manual install instructions

Download Ollama

Also works in WSL2! ☺

macOS



Windows

Install with one command:

```
curl -fsSL https://ollama.com/install.sh | sh
```

View script source • Manual install instructions

llama3.2

Meta's Llama 3.2 goes small with 1B and 3B models.

tools 1b 3b

1.9M Pulls Updated 4 weeks ago

3b 63 Tags

```
ollama run llama3.2
```

Updated 4 weeks ago		a80c4f17acd5 · 2.0GB
model	arch **llama** · parameters **3.21B** · quantization **Q4_K_M**	2.0GB
params	{ "stop": ["<|start_header_id|>", "<|end_header_id|>", "<|eot_i…	96B
template	<|start_header_id|>system<|end_header_id|> Cutting Knowledge Dat…	1.4kB
license	**Llama 3.2** **Acceptable Use Policy** Meta is committed to pro…	6.0kB
license	LLAMA 3.2 COMMUNITY LICENSE AGREEMENT Llama 3.2 Version Release …	7.7kB

llama3.2

Meta's Llama 3.2 goes small with 1B and 3B models.

tools 1b 3b

1.9M Pulls Updated 4 weeks ago

Watch out for model hash/ID!

63 Tags

latest
a80c4f17acd5 • 2.0GB • 4 weeks ago

1b
baf6a787fdff • 1.3GB • 4 weeks ago

3b
a80c4f17acd5 • 2.0GB • 4 weeks ago

1b-instruct-fp16
2887c3d03e74 • 2.5GB • 4 weeks ago

1b-instruct-q2_K
3718017cfd4e • 581MB • 4 weeks ago

1b-instruct-q3_K_L
1a709e91d2fb • 733MB • 4 weeks ago

1b-instruct-q3_K_M
8459ea7be88f • 691MB • 4 weeks ago

1b-instruct-q3_K_S
109ea9f8f55f • 642MB • 4 weeks ago

3b-instruct-q3_K_L
adcbc7b3c10e • 1.8GB • 4 weeks ago

3b-instruct-q3_K_M
fea4b4677930 • 1.7GB • 4 weeks ago

3b-instruct-q3_K_S
860e23062c32 • 1.5GB • 4 weeks ago

3b-instruct-q4_0
9b9453afbdd6 • 1.9GB • 4 weeks ago

3b-instruct-q4_1
c910c5139ab6 • 2.1GB • 4 weeks ago

3b-instruct-q4_K_M
a80c4f17acd5 • 2.0GB • 4 weeks ago

3b-instruct-q4_K_S
80f2089878c9 • 1.9GB • 4 weeks ago

3b-instruct-q5_0
fa2b62a5f96d • 2.3GB • 4 weeks ago

3b-instruct-q5_1
0452394ac7c9 • 2.4GB • 4 weeks ago

3b-instruct-q5_K_M
7709c7357e6d • 2.3GB • 4 weeks ago

3b-instruct-q5_K_S
97ef2f873c2c • 2.3GB • 4 weeks ago

AutoGen + Ollama

```
from autogen import AssistantAgent, UserProxyAgent

config_list = [
  {
    "model": "codellama",
    "base_url": "http://localhost:11434/v1",
    "api_key": "ollama",
  }
]

assistant = AssistantAgent("assistant", llm_config={"config_list":
config_list})

user_proxy = UserProxyAgent("user_proxy", code_execution_config={"work_dir":
"coding", "use_docker": False})
user_proxy.initiate_chat(assistant, message="Plot a chart of NVDA and TESLA
stock price change YTD.")
```

AutoGen Studio

AutoGen Studio
Build Multi-Agent Apps

Build | Playground | Gallery

Guest User
guestuser

Workflow

Select or create an agent workflow.

General Agent Workflow

Create new workflows here

Sessions

Create a new session or select an existing session to view chat.

ada9e3bf-4079-4f66-8c8e-6a3 ...
just now
delete publish

45d4da7e-f9aa-4609-a408-984 ...
28 minutes ago
delete publish

680d8f0c-3e8f-4bb2-8c06-b54 ...
19 hours ago
delete publish

+ New

close sidebar

USER

create a 4 page pdf brochure on coffee from different parts of the world with some description of origins. E.g Ethiopian coffee may be in a glass on a table with a lush green forest in the background.

AGENT

The PDF brochure titled "Coffee_Brochure.pdf" has been successfully created. It includes images and descriptions of coffee from Ethiopia, Colombia, Brazil, and Vietnam, assembled into a 4-page document. Your brochure on coffee from different parts of the world is now ready. TERMINATE

Agent Messages (10 messages) | 2 mins 39 secs

Results (8 files)

Coffee_Brochure.pdf

PDF

77d9370c-1fe1-4658-a79a-ab7890c93a6b.png

f7a649b1-ce4a-4d5f-91a3-0085b2f574b6.png

a24eea2a-8d7c-4855-a1a8-953b08b0bf41.png

6d3319d6-8041-49e1-acb5-439f084440c5.png

generate_more_images.py

create_pdf_brochure.py

generate_images.py

Blank slate? Try one of the example prompts below

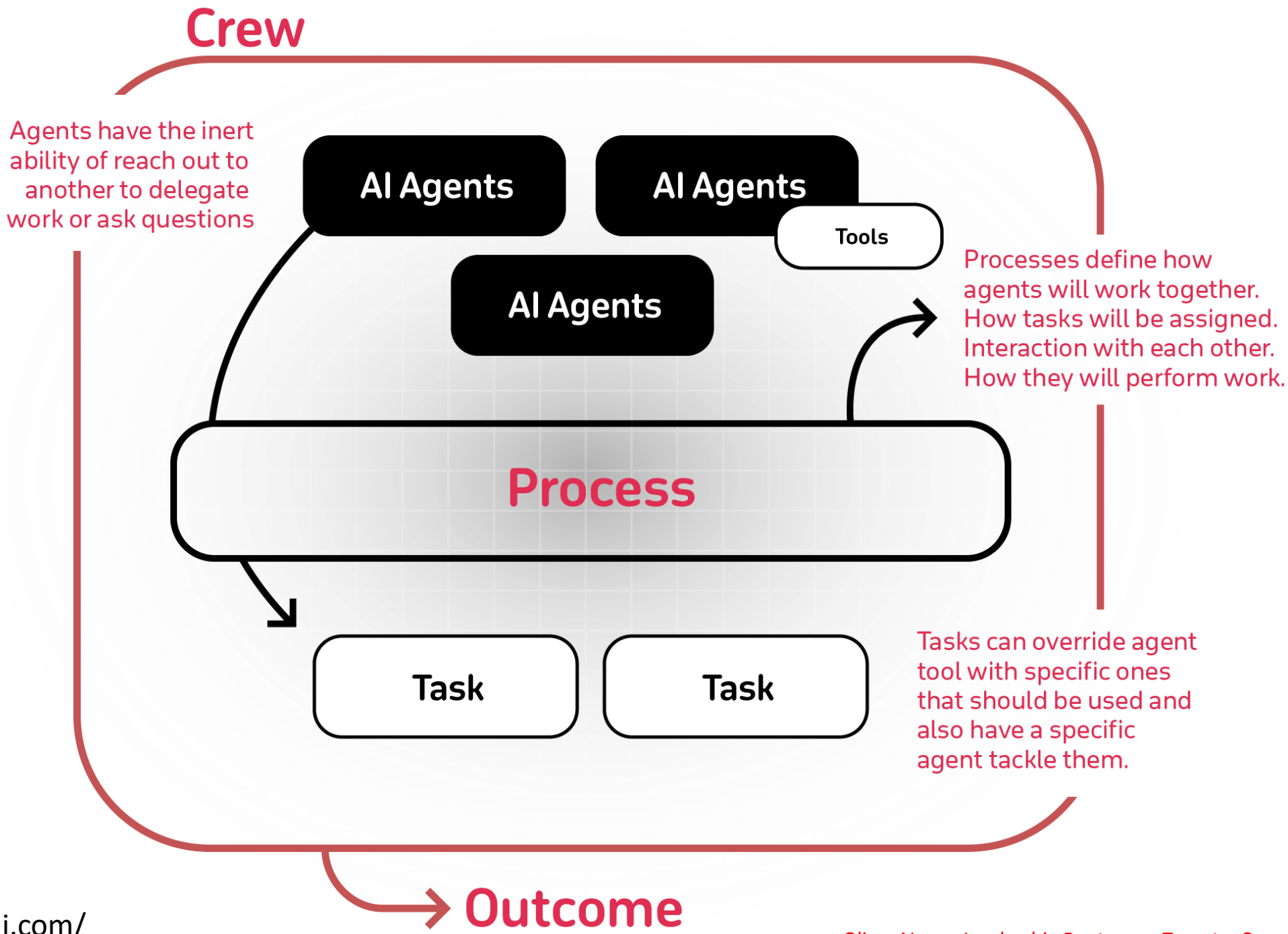
Stock Price | Sine Wave | Markdown | Paint

Maintained by the AutoGen Team.

Overview of the session

- ❑ Introduction
- ❑ Deep dive into Microsoft AutoGen with examples
- ❑ Creating an AutoGen playground for experimentation
- ❑ **Other agentic development platforms**
- ❑ Standardization efforts for FM-powered Agents
- ❑ Beyond this presentation


crewai



https://www.crewai.com/

agents.yaml

```yaml
# src/latest_ai_development/config/agents.yaml
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
    developments in {topic}. Known for your ability to find the most relevant
    information and present it in a clear and concise manner.

reporting_analyst:
  role: >
    {topic} Reporting Analyst
  goal: >
    Create detailed reports based on {topic} data analysis and research findings
  backstory: >
    You're a meticulous analyst with a keen eye for detail. You're known for
    your ability to turn complex data into clear and concise reports, making
    it easy for others to understand and act on the information you provide.
```

tasks.yaml

```yaml
# src/latest_ai_development/config/tasks.yaml
research_task:
  description: >
    Conduct a thorough research about {topic}
    Make sure you find any interesting and relevant information given
    the current year is 2024.
  expected_output: >
    A list with 10 bullet points of the most relevant information about {topic}
  agent: researcher

reporting_task:
  description: >
    Review the context you got and expand each topic into a full section for a report.
    Make sure the report is detailed and contains any and all relevant information.
  expected_output: >
    A fully fledge reports with the mains topics, each with a full section of information
    Formatted as markdown without '```'
  agent: reporting_analyst
  output_file: report.md
```

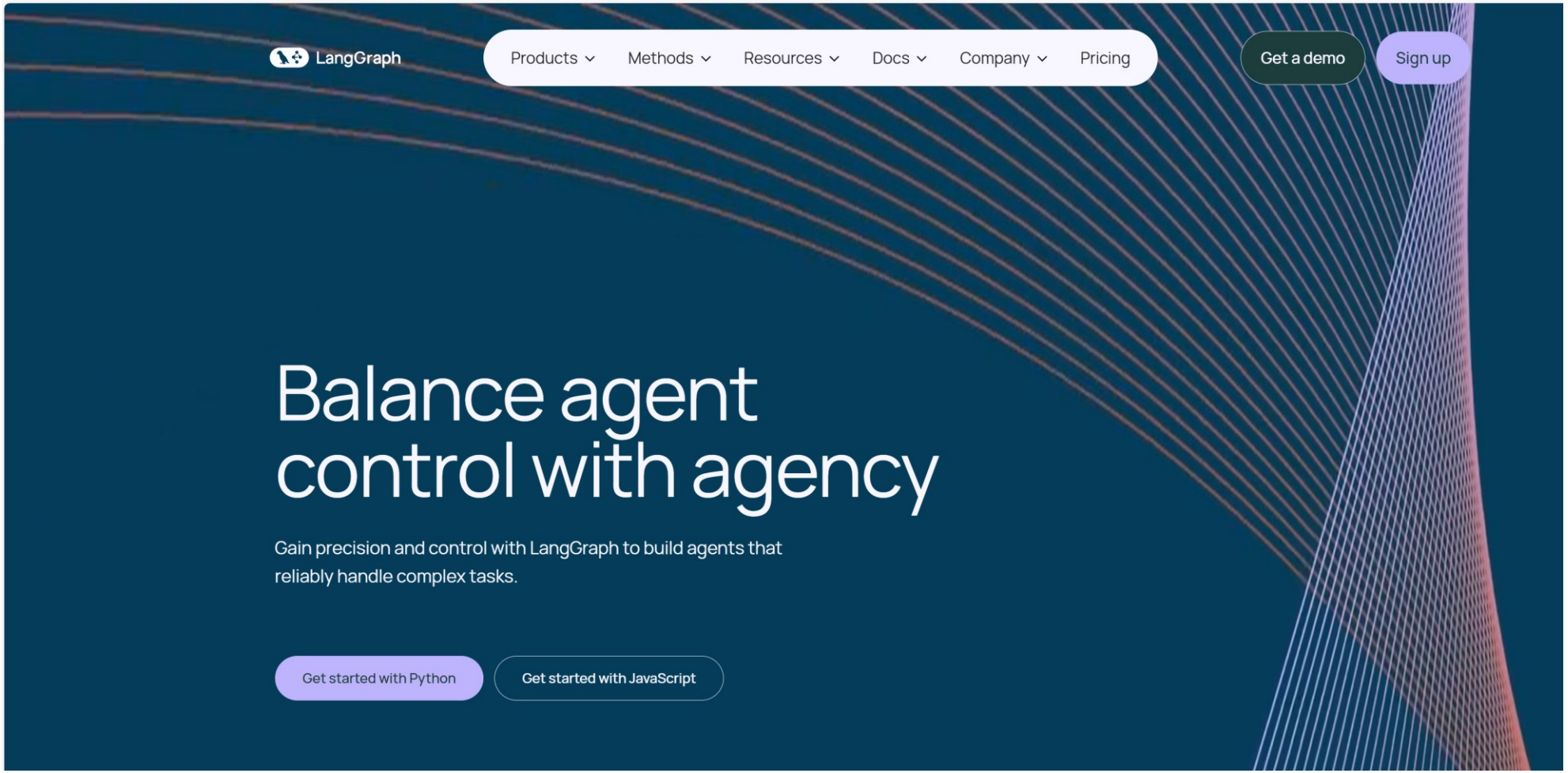
https://github.com/openai/swarm

Swarm (experimental, educational)

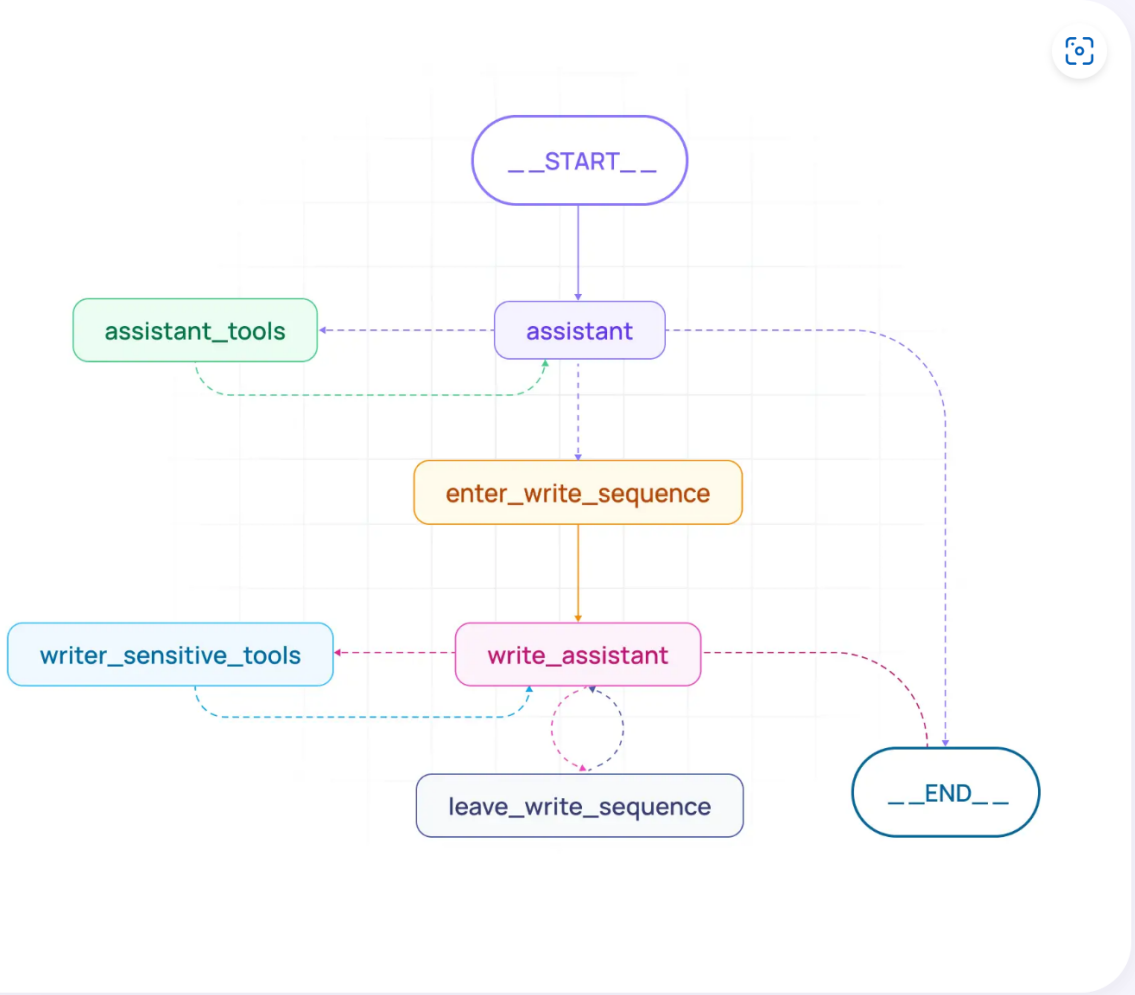
An educational framework exploring ergonomic, lightweight multi-agent orchestration.

> ⚠ **Warning**
>
> Swarm is currently an experimental sample framework intended to explore ergonomic interfaces for multi-agent systems. It is not intended to be used in production, and therefore has no official support. (This also means we will not be reviewing PRs or issues!)
>
> The primary goal of Swarm is to showcase the handoff & routines patterns explored in the Orchestrating Agents: Handoffs & Routines cookbook. It is not meant as a standalone library, and is primarily for educational purposes.

LangGraph



https://www.langchain.com/langgraph



Controllable cognitive architecture for any task

LangGraph's flexible framework supports diverse control flows – single agent, multi-agent, hierarchical, sequential – and robustly handles realistic, complex scenarios.

Ensure reliability with easy-to-add moderation and quality loops that prevent agents from veering off course.

Use LangGraph Platform to templatize your cognitive architecture so that tools, prompts, and models are easily configurable with LangGraph Platform Assistants.

See the docs ↗

	AutoGen	CrewAI	Open AI Swarm	Langgraph
Popularity	**32.4k stars**	20.5k stars	15k stars	6.4k stars
Primary Use Case	Autonomous multi-agent systems	Task automation and workforce optimization	Collaborative agent orchestration	Dynamic conversational agents
Platform Focus	Autonomous agent interaction	Workforce task allocation	Swarm intelligence and collaboration	NLP model interactions and flows
Collaboration Model	Multi-agent autonomous synergy	Task assignment (human-AI blend)	Swarm-based, collaborative agents	Node-based, agent-to-agent flows
Customization Level	Moderate; code-based custom workflows	Moderate; predefined workflows	High; modular swarm architecture	High; tailored conversation flows
Deployment	Cloud and edge deployment options	Cloud and on-premise	Cloud-native only	Cloud-based
User Interface	Comprehensive UI for agent workflows	Dashboard-focused for task management	Modular, customizable dashboards	Visual, node-based graphing interface

Overview of the session

- ❑ Introduction
- ❑ Deep dive into Microsoft AutoGen with examples
- ❑ Creating an AutoGen playground for experimentation
- ❑ Other agentic development platforms
- ❑ **Standardization efforts for FM-powered Agents**
- ❑ Beyond this presentation

A standard is needed to enable interoperability

- Developers are **building agents in their own way** (ad-hoc) or using different frameworks
- The **lack of a common/unified interface** for agents creates several problems:
  - Hard to **compare** (e.g., benchmark) agents
  - Hard to **reuse** agents
  - Hard to **develop tools** that would work with any agent out-of-the-box
- Due to the increased adoption of agentic cognitive architectures, **agent interoperability will become a key challenge**

P3394

Standard for Large Language Model Agent Interface

Active PAR

This standard defines natural language interfaces that facilitate communication between Large Language Model (LLM) applications, agents, and human users. The standard defines a set of protocols and guidelines that enable applications and agents to effectively communicate with LLM enabled Agents. Thereby, the standard enables seamless interactions between multiple applications and agents. The standard covers a wide range of aspects related to LLM usage and application, including but not limited to API syntax and semantics, voice and text format, conversational flow, prompt engineering integration, LLM chain of thoughts integration, and API endpoint configuration, authentication and authorization for LLM plugins.

Sponsor Committee	C/AISC - Artificial Intelligence Standards Committee
Status	Active PAR
PAR Approval	2023-09-21 >

WORKING GROUP DETAILS

Society	IEEE Computer Society Learn More About IEEE Computer Society >
Sponsor Committee	C/AISC - Artificial Intelligence Standards Committee
Working Group	LLM-AAI - Large Language Model Application and Agent Interface
IEEE Program Manager	Christy Bahn Contact Christy Bahn >
Working Group Chair	Richard Tong

Agent Protocol by AI Engineer Foundation

- **Goal: Develop a unified protocol/interface that is as simple as possible**
- The Agent Protocol is an API specification (OpenAPI specification v3) and thus technology agnostic
  - List of endpoints that the agent should expose with predefined response schema
- The three base objects of the protocol are **Task**, **Step**, and **Artifact**
  - A **Task** denotes one specific goal for the agent, which can be very specific or very broad
  - A **Step** is a single action that the agent should perform. Each step is triggered by calling the **step** endpoint of the agent.
  - An **Artifact** is a file that the agent has worked with.
- The protocol has two main endpoints:
    - **/ap/v1/agent/tasks** [POST] - This endpoint is used to create a new task for the agent.
    - **/ap/v1/agent/tasks/{task_id}/steps** [POST] - This endpoint is used to trigger next step of the task.

Overview of the session

- ❑ Introduction
- ❑ Deep dive into Microsoft AutoGen with examples
- ❑ Creating an AutoGen playground for experimentation
- ❑ Other agentic development platforms
- ❑ Standardization efforts for FM-powered Agents
- ❑ **Beyond this presentation**

Where to go from here?

- **Productionizing Agentic FMware is really hard**
  - Make sure you check Ahmed/Gopi's talk about the challenges of productionizing AIware
- **AgentOps and (Semantic) Observability are crucial!**
  - Make sure you check Ben's presentation about AIware Observability on Day 6 (11:00-11:45am)
- **About Tools…**
  - Check out other memory frameworks (e.g., Zep)
  - AutoGen 0.4 experimental will be released soon and has a nicer API
  - Several cool videos and courses on Youtube / Coursera
  - FMArts hands-on
- **Autonomous Software Engineers is an interesting use case**
  - Agentless (why agents?)
  - Aide (agents everywhere!)
  - SWE-bench