AutoGen

Master Cheatsheet 2025

Multi-Agent AI Framework | Microsoft | Complete Guide

■ What is AutoGen?

AutoGen = Microsoft's open-source framework for building multi-agent AI applications where multiple AI agents collaborate, converse, and execute code to solve complex tasks.

Feature	Description
Multi-Agent	Multiple AI agents working together
Conversational	Agents communicate via natural language
Code Execution	Built-in safe code execution (Docker/Local)
Human-in-Loop	Human feedback at any conversation point
Tool Use	Agents can use custom functions/tools
Flexible LLMs	OpenAI, Azure, Anthropic, Ollama, etc.
Group Chat	Multiple agents in one conversation
Customizable	Create custom agent types

■■ Installation & Setup

Package	Command	Purpose
Basic	pip install pyautogen	Core AutoGen
With Tools	pip install "pyautogen[tools]"	Tool/function calling
With RAG	pip install "pyautogen[retrievechat]"	RAG capabilities
Full Install	pip install "pyautogen[all]"	All features
AG2 (New)	pip install ag2	AutoGen 0.4+ (AG2)
AG2 OpenAI	pip install "ag2[openai]"	AG2 with OpenAI

■ LLM Configuration

```
# OAI_CONFIG_LIST file or dict
llm_config = {
'config_list': [
{
'model': 'gpt-4o',
'api_key': 'your-openai-key', # or use env: OPENAI_API_KEY
},
{
'model': 'gpt-4o-mini',
'api_key': 'your-openai-key',
},
],
'temperature': 0,
'timeout': 120,
}

# Or load from file/env
config_list = autogen.config_list_from_json('OAI_CONFIG_LIST')
config_list = autogen.config_list_from_dotenv('.env')
```

■ Agent Types

Agent Type	Description	Use Case
AssistantAgent	AI-powered agent with LLM	Task solving, coding, reasoning
UserProxyAgent	Represents human user	Execute code, human feedback
ConversableAgent	Base class for all agents	Custom agent creation
GroupChatManager	Manages multi-agent chat	Coordinate group conversations
RetrieveAssistantAgent	RAG-enabled assistant	Document Q&A
RetrieveUserProxyAgent	RAG-enabled user proxy	Retrieve & process docs
GPTAssistantAgent	OpenAI Assistants API	Use OpenAI Assistants
MultimodalAgent	Vision + text agent	Image understanding

```
import autogen

# Assistant Agent (AI that helps solve tasks)
assistant = autogen.AssistantAgent(
name='assistant',
llm_config=llm_config,
system_message='You are a helpful AI assistant.'
)

# User Proxy Agent (represents human, can execute code)
user_proxy = autogen.UserProxyAgent(
name='user_proxy',
human_input_mode='NEVER', # NEVER, ALWAYS, TERMINATE
max_consecutive_auto_reply=10,
is_termination_msg=lambda x: 'TERMINATE' in x.get('content', ''),
code_execution_config={'work_dir': 'coding', 'use_docker': False}
)

# Start conversation
user_proxy.initiate_chat(assistant, message='Write a Python function to calculate factorial')
```

■ Conversation Patterns

Pattern	Description	Code

Two-Agent Chat	Simple A to B conversation	agent_a.initiate_chat(agent_b, message="...")
Sequential Chat	Chain of conversations	initiate_chats([{...}, {...}])
Group Chat	Multiple agents together	GroupChat + GroupChatManager
Nested Chat	Sub-conversations within chat	register_nested_chats()
Async Chat	Non-blocking conversations	a]_initiate_chat()

■ Human Input Modes

Mode	Behavior
NEVER	Never ask for human input (fully autonomous)
ALWAYS	Always ask for human input before responding
TERMINATE	Ask for input only when termination condition met

■ Termination Conditions

```
# Custom termination function
def is_termination_msg(message):
content = message.get('content', '')
return content and ('TERMINATE' in content or 'TASK COMPLETE' in content)

user_proxy = autogen.UserProxyAgent(
name='user',
is_termination_msg=is_termination_msg,
max_consecutive_auto_reply=5, # Max auto replies before stopping
)
```

■ Code Execution

Setting	Options	Description
use_docker	True / False / "image_name"	Run code in Docker container
work_dir	Directory path	Working directory for code files
timeout	Seconds (int)	Max execution time
last_n_messages	int or "auto"	Messages to scan for code

```
# Code execution configuration
code_execution_config = {
'work_dir': 'coding', # Directory to save/run code
'use_docker': False, # Set True for safety in production
'timeout': 60, # Timeout in seconds
'last_n_messages': 3, # Check last 3 messages for code
}

# Disable code execution entirely
code_execution_config = False

# Docker execution (safer)
code_execution_config = {
'use_docker': 'python:3.11-slim',
'work_dir': 'coding'
}
```

■ Tools & Function Calling

■ *Register custom functions that agents can call during conversation*

```
from typing import Annotated

# Define a tool function
def get_weather(
city: Annotated[str, 'City name to get weather for']
) -> str:
'''Get current weather for a city'''
return f'Weather in {city}: 25C, Sunny'

def calculate(
expression: Annotated[str, 'Math expression to evaluate']
) -> str:
'''Calculate a math expression'''
return str(eval(expression))

# Register tools with agents
assistant = autogen.AssistantAgent('assistant', llm_config=llm_config)
user_proxy = autogen.UserProxyAgent('user', code_execution_config=False)

# Register for LLM (assistant can suggest calling it)
assistant.register_for_llm(name='get_weather', description='Get weather')(get_weather)
assistant.register_for_llm(name='calculate', description='Calculate math')(calculate)

# Register for execution (user_proxy will execute it)
user_proxy.register_for_execution(name='get_weather')(get_weather)
user_proxy.register_for_execution(name='calculate')(calculate)
```

■ Decorator Approach (Simpler)

```
from autogen import register_function

@register_function(caller=assistant, executor=user_proxy)
def search_web(query: str) -> str:
'''Search the web for information'''
# Your search implementation
return f'Search results for: {query}'
```

■ Group Chat (Multi-Agent)

Component	Purpose
GroupChat	Container for multiple agents + conversation rules
GroupChatManager	Orchestrates the conversation, decides who speaks next
speaker_selection_method	How to choose next speaker: auto, round_robin, random, manual
max_round	Maximum conversation turns
admin_name	Name of admin agent (optional)
allow_repeat_speaker	Can same agent speak twice in a row

```
# Create specialized agents
coder = autogen.AssistantAgent(
name='Coder',
system_message='You are a Python developer. Write clean code.',
llm_config=llm_config
)

reviewer = autogen.AssistantAgent(
name='Reviewer',
system_message='You review code for bugs and improvements.',
llm_config=llm_config
)

tester = autogen.AssistantAgent(
name='Tester',
system_message='You write test cases for code.',
llm_config=llm_config
)

user_proxy = autogen.UserProxyAgent('Admin', human_input_mode='NEVER')

# Create group chat
groupchat = autogen.GroupChat(
agents=[user_proxy, coder, reviewer, tester],
messages=[],
max_round=12,
speaker_selection_method='auto' # LLM decides who speaks
)

manager = autogen.GroupChatManager(groupchat=groupchat, llm_config=llm_config)

# Start group conversation
user_proxy.initiate_chat(manager, message='Create a calculator with tests')
```

■ Speaker Selection Methods

Method	Description
auto	LLM decides who should speak next (default)
round_robin	Agents speak in order, one after another
random	Randomly select next speaker
manual	Human selects next speaker

■ RAG (Retrieval Augmented Generation)

■ Install: pip install "pyautogen[retrievechat]"

```
from autogen.agentchat.contrib.retrieve_assistant_agent import RetrieveAssistantAgent
from autogen.agentchat.contrib.retrieve_user_proxy_agent import RetrieveUserProxyAgent

# RAG Assistant
rag_assistant = RetrieveAssistantAgent(
name='rag_assistant',
system_message='Answer questions based on the retrieved context.',
llm_config=llm_config
)

# RAG User Proxy with document retrieval
rag_proxy = RetrieveUserProxyAgent(
name='rag_proxy',
human_input_mode='NEVER',
retrieve_config={
'task': 'qa', # 'qa', 'code', 'default'
'docs_path': ['./documents', 'https://example.com/doc.pdf'],
'chunk_token_size': 1000,
'model': 'gpt-4o',
'collection_name': 'my_docs',
'get_or_create': True,
'embedding_model': 'text-embedding-3-small',
}
)

# Query your documents
rag_proxy.initiate_chat(rag_assistant, message='What does the document say about X?')
```

Config	Description
task	Task type: qa, code, default
docs_path	List of file paths or URLs
chunk_token_size	Size of document chunks
collection_name	Vector DB collection name
embedding_model	Model for embeddings
get_or_create	Reuse existing collection
customized_prompt	Custom RAG prompt template

■ LLM Provider Configurations

Provider	Configuration
OpenAI	{'model': 'gpt-4o', 'api_key': 'sk-...'}
Azure OpenAI	{'model': 'gpt-4', 'api_key': '...', 'base_url': 'https://xxx.openai.azure.com', 'api_type': 'azure', 'api_version': '2024-02-01'}
Anthropic	{'model': 'claude-sonnet-4-20250514', 'api_key': '...', 'api_type': 'anthropic'}
Ollama	{'model': 'llama3.2', 'base_url': 'http://localhost:11434/v1', 'api_key': 'ollama'}
Groq	{'model': 'llama-3.3-70b-versatile', 'api_key': '...', 'base_url': 'https://api.groq.com/openai/v1'}
Together	{'model': 'meta-llama/Llama-3-70b', 'api_key': '...', 'base_url': 'https://api.together.xyz/v1'}
Mistral	{'model': 'mistral-large-latest', 'api_key': '...', 'base_url': 'https://api.mistral.ai/v1'}

■ Advanced Patterns

Pattern	Description	Use Case
Sequential Chats	Chain multiple conversations	Multi-step workflows
Nested Chats	Sub-conversations triggered by events	Complex decision trees
Custom Agents	Extend ConversableAgent	Specialized behaviors
Teachable Agent	Agent that learns from feedback	Improving over time
Society of Mind	Hierarchical agent teams	Large complex tasks
AutoBuild	Auto-generate agent teams	Dynamic team creation
StateFlow	FSM-based agent workflows	Structured conversations

```
# Sequential Chats - Chain multiple conversations
chat_results = user_proxy.initiate_chats([
{
'recipient': researcher,
'message': 'Research topic X',
'max_turns': 3,
'summary_method': 'last_msg'
},
{
'recipient': writer,
'message': 'Write article based on research',
'max_turns': 3,
'summary_method': 'reflection_with_llm'
},
{
'recipient': editor,
'message': 'Edit and polish the article',
'max_turns': 2
}
])
```

■ AG2 (AutoGen 0.4+)

■ AG2 is the new version with improved architecture, better async support, and cleaner APIs

```
# AG2 Installation
# pip install ag2[openai]

from ag2 import AssistantAgent, UserProxyAgent
from ag2.oai import OpenAIChatCompletionClient

# Create client
client = OpenAIChatCompletionClient(model='gpt-4o')

# Create agents (similar API)
assistant = AssistantAgent(
name='assistant',
model_client=client,
system_message='You are helpful.'
)

user = UserProxyAgent(name='user')

# Async by default
await user.initiate_chat(assistant, message='Hello!')
```

■ Best Practices

Category	Best Practice

System Messages	Write clear, specific system messages for each agent role
Termination	Always define clear termination conditions
Code Execution	Use Docker in production for safe code execution
Temperature	Use temperature=0 for deterministic, reproducible results
Max Turns	Set max_consecutive_auto_reply to prevent infinite loops
Error Handling	Wrap initiate_chat in try-except blocks
Logging	Enable logging to debug agent conversations
Cost Control	Use cheaper models (gpt-4o-mini) for simple agents
Testing	Test with human_input_mode=ALWAYS first
Functions	Keep tool functions simple and well-documented
Group Chat	Limit group chat to 3-5 agents for best results
Context	Summarize long conversations to manage context length

■ Common Use Cases

Use Case	Agents Needed	Pattern
Code Generation	Coder + UserProxy	Two-agent with code execution
Code Review	Coder + Reviewer + Tester	Group Chat
Research Assistant	Researcher + Writer + Editor	Sequential Chat
Customer Support	Support Agent + RAG Proxy	RAG Chat
Data Analysis	Analyst + Coder + UserProxy	Group + Code Execution
Content Creation	Writer + Editor + Fact Checker	Group Chat
Task Planning	Planner + Executor + Critic	Group Chat
Tutoring	Teacher + Student (Human)	Human-in-loop

■ Quick Reference

Task	Code
Start Chat	agent_a.initiate_chat(agent_b, message="...")
Get Last Message	agent.last_message()
Get Chat History	agent.chat_messages[other_agent]
Clear History	agent.clear_history()
Reset Agent	agent.reset()
Register Function	@register_function(caller=a, executor=b)
Generate Reply	agent.generate_reply(messages=[...])
Send Message	agent.send(message, recipient)

Created by Manoj | The AI Dude Tamil ■

YouTube: The AI Dude Tamil | Master AI, Automation & Prompt Engineering

■ Build powerful multi-agent AI with AutoGen!