DocCompiler.ai

USER GUIDE

Autogen User Guide

Installation, Usage & Quick Reference

A Cyber Panda Solutions LLC product

Version 3.1 2026-05-13 @ 027ecf0 Generated by DocCompiler.ai

Always-updated version at doccompiler.ai/microsoft/autogen

Source code analysis by Claude Opus 4.6 | 1M token context

About This Document

What Is This?

This is a comprehensive, source-code-verified **User Guide** for **Autogen**, generated by DocCompiler.ai using Claude Opus 4.6 with 1M-token context. Every command, configuration option, and behavior documented here was verified against the actual source code.

The Definitive Reference

This document is designed to be the single source of truth for Autogen. Rather than piecing together scattered README fragments, outdated wiki pages, forum posts, and version-specific screenshots, you can rely on this as a verified, structured reference backed by automated code review.

Optimized for AI Assistants

This document is structured for consumption by AI assistants. Feed this PDF into your preferred LLM for accurate, grounded answers about Autogen — instead of having the model guess or hallucinate from incomplete context.

Version & Updates

Generated on **2026-05-13** (version **3.1**). This document reflects the source code at commit `027ecf0`. The always-updated version is permanently available at **doccompiler.ai/microsoft/autogen**. Do not print or download for offline use — this document is updated when the source code changes.

Our Mission

DocCompiler.ai aims to be the definitive wiki for the world's most important open-source projects. Every document is overkill by design: exhaustive, fact-checked, and built to replace the scattered tribal knowledge that slows teams down.

Table of Contents

AutoGen User Guide

1. Quick Start

AutoGen is Microsoft's open-source framework for building multi-agent AI applications. It provides a modular, event-driven architecture where specialized AI agents collaborate through conversations, tool use, and code execution to solve complex tasks.

Key features:

- Build multi-agent systems with conversational AI agents in Python or .NET
- Pre-built agent types: `AssistantAgent`, `UserProxyAgent`, and custom `RoutedAgent`
- Group chat patterns: `RoundRobinGroupChat`, `SelectorGroupChat`, and graph-based workflows
- Pluggable model backends: OpenAI, Anthropic, Azure AI, Gemini, Ollama, Mistral, LLama.cpp
- Built-in code execution (local, Docker, Jupyter, Azure)
- MCP (Model Context Protocol) tool integration
- AutoGen Studio: a visual web UI for building and testing agent teams
- Distributed runtimes via gRPC for cross-process and cross-language agent communication

Prerequisites: Python 3.10+ or .NET 8.0+

Install and run your first agent (Python):

bash

```
pip install autogen-agentchat autogen-ext[openai]
```

python

```
import asyncio
from autogen_agentchat.agents import AssistantAgent
from autogen_ext.models.openai import OpenAIChatCompletionClient

async def main():
    model_client = OpenAIChatCompletionClient(model="gpt-4o")
    agent = AssistantAgent("assistant", model_client=model_client)
    response = await agent.on_messages(
        [{"content": "What is AutoGen?", "source": "user"}], cancellation_token=None
    )
    print(response.chat_message.content)

asyncio.run(main())
```

> **Tip**
>
> Set the `OPENAI_API_KEY` environment variable before running. For other providers, see the Configuration section.

What to try next:

- Set up a multi-agent group chat (see Core Usage)
- Launch AutoGen Studio for a visual experience (see Common Tasks)
- Run agents across processes with gRPC (see the Admin Guide for distributed runtime details)

2. Installation

System Requirements

Requirement	Python SDK	.NET SDK
Runtime	Python 3.10+	.NET 8.0+
OS	Windows, macOS, Linux	Windows, macOS, Linux
Package manager	pip or uv	NuGet
Optional	Docker (for sandboxed code execution)	Docker, .NET Aspire

Python Installation

The Python SDK is organized as a family of packages. Install only what you need:

Package	Purpose	Install
`autogen-core`	Core runtime, agent base classes, messaging	`pip install autogen-core`
`autogen-agentchat`	High-level chat agents, teams, termination conditions	`pip install autogen-agentchat`
`autogen-ext`	Extensions: model clients, code executors, tools, memory	`pip install autogen-ext`
`pyautogen`	Legacy compatibility shim (v0.2 API)	`pip install pyautogen`

Install with model provider extras:

bash

```
# OpenAI / Azure OpenAI
pip install autogen-ext[openai]

# Anthropic (Claude)
pip install autogen-ext[anthropic]

# Azure AI Inference
pip install autogen-ext[azure]

# Ollama (local models)
pip install autogen-ext[ollama]

# Multiple providers at once
pip install autogen-ext[openai,anthropic,azure]

# Magentic-One (multi-agent web browsing system)
```

```
pip install autogen-ext[openai,magentic-one]
```

> **Important**
>
> Each model provider extra installs its own SDK dependency. You must also set the corresponding API key environment variable (e.g., `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`).

Install from source (development):

bash

```
git clone https://github.com/microsoft/autogen.git
cd autogen/python
uv sync
source .venv/bin/activate
```

.NET Installation

Add the AutoGen NuGet packages to your project:

xml

```
&lt;ItemGroup&gt;
    &lt;PackageReference Include="Microsoft.AutoGen.Core" /&gt;
    &lt;PackageReference Include="AutoGen.OpenAI" /&gt;
&lt;/ItemGroup&gt;
```

Available .NET packages:

Package	Purpose
`Microsoft.AutoGen.Core`	Core runtime, agent hosting, subscriptions
`Microsoft.AutoGen.Contracts`	Shared interfaces and message types
`AutoGen.OpenAI`	OpenAI model client with middleware
`AutoGen.Anthropic`	Anthropic model client with middleware
`AutoGen.Gemini`	Google Gemini model client
`AutoGen.Mistral`	Mistral model client
`AutoGen.SemanticKernel`	Semantic Kernel integration
`AutoGen.AzureAIInference`	Azure AI model client
`AutoGen.Ollama`	Ollama local model client
`AutoGen.LMStudio`	LM Studio local server client
`AutoGen.SourceGenerator`	Type-safe function definition generation
`AutoGen.DotnetInteractive`	.NET Interactive code execution

AutoGen Studio Installation

AutoGen Studio provides a web-based UI for visually building and testing agent teams.

bash

```
pip install autogenstudio
```

Verify installation:

bash

```
autogenstudio version
```

text

```
AutoGen Studio  CLI version: 0.x.x
```

AgBench Installation

AgBench is AutoGen's benchmarking tool for evaluating agent performance.

bash

```
cd autogen/python/packages/agbench
pip install -e .
```

Verify:

bash

```
autogenbench --version
```

Post-Install Verification (Python)

bash

```
python -c "from autogen_agentchat.agents import AssistantAgent; print('OK')"
```

text

```
OK
```

> **Warning**
>
> The `pyautogen` package provides backwards compatibility with AutoGen v0.2 APIs. For new projects, use the `autogen-core` and `autogen-agentchat` packages directly.

Install Troubleshooting

Problem	Solution
`ModuleNotFoundError: autogen_agentchat`	Install the correct package: `pip install autogen-agentchat`
`ModuleNotFoundError: autogen_ext.models.openai`	Install the extras: `pip install autogen-ext[openai]`
`OPENAI_API_KEY not set`	Export the key: `export OPENAI_API_KEY=sk-...`
Protobuf import errors (gRPC runtime)	Run `python fixup_generated_files.py` from the `python/` directory

.NET Aspire dashboard not showing	Ensure .NET 8.0+ SDK is installed and run from the `.AppHost` project
Docker code executor fails	Verify Docker Desktop is running and has WSL integration enabled
`pip install` conflicts	Use a virtual environment: `python -m venv .venv && source .venv/bin/activate`
`autogenstudio` command not found	Ensure pip installed scripts are on your PATH
WebSocket deprecation warnings	These are harmless; AutoGen Studio suppresses them automatically

3. Configuration

Model Client Configuration

AutoGen uses a unified `ChatCompletionClient` interface. Each model provider has its own configuration class.

OpenAI / Azure OpenAI:

```python
from autogen_ext.models.openai import OpenAIChatCompletionClient

# Standard OpenAI
client = OpenAIChatCompletionClient(model="gpt-4o")

# Azure OpenAI
from autogen_ext.models.openai import AzureOpenAIChatCompletionClient
client = AzureOpenAIChatCompletionClient(
    model="gpt-4o",
    azure_deployment="my-deployment",
    azure_endpoint="https://my-resource.openai.azure.com/",
    api_key="your-azure-key",
)
```

Anthropic (Claude):

```python
from autogen_ext.models.anthropic import AnthropicChatCompletionClient

client = AnthropicChatCompletionClient(model="claude-3-5-sonnet-20241022")
```

Ollama (local models):

```python
from autogen_ext.models.ollama import OllamaChatCompletionClient

client = OllamaChatCompletionClient(model="llama3.1")
```

> **Tip**
> You can load model clients from JSON or YAML configuration files using `ChatCompletionClient.load_component(config_data)`. This keeps API keys and model settings separate from code.

Configuration file example (`model_config.yml`):

```yaml
provider: autogen_ext.models.openai.OpenAIChatCompletionClient
config:
  model: gpt-4o
  api_key: ${OPENAI_API_KEY}
```

Environment Variables

Variable	Provider	Purpose
`OPENAI_API_KEY`	OpenAI	API authentication
`ANTHROPIC_API_KEY`	Anthropic	API authentication
`AZURE_OPENAI_API_KEY`	Azure OpenAI	API authentication
`AZURE_OPENAI_ENDPOINT`	Azure OpenAI	Service endpoint URL
`AZURE_OPENAI_DEPLOYMENT_NAME`	Azure OpenAI	Deployment name
`AZURE_AI_INFERENCE_ENDPOINT`	Azure AI	Inference endpoint
`AZURE_AI_INFERENCE_API_KEY`	Azure AI	Inference API key
`GEMINI_API_KEY`	Google Gemini	API authentication
`HF_TOKEN`	Hugging Face	Model access token
`GATSBY_API_URL`	AutoGen Studio	Backend API URL for frontend

Model Capabilities

AutoGen tracks model capabilities through the `ModelInfo` interface:

Field	Type	Description
`vision`	`bool`	Whether the model supports image input
`function_calling`	`bool`	Whether the model supports tool/function calling
`json_output`	`bool`	Whether the model supports JSON mode
`structured_output`	`bool`	Whether the model supports structured output schemas
`family`	`str`	Model family identifier (e.g., `gpt-4o`, `claude-3-5-sonnet`)

`multiple_system_messages`	`bool`	Whether multiple non-consecutive system messages are supported

> **Note**
> Model capabilities are auto-detected from the model name. Override them with the `model_info` parameter if your provider uses non-standard model names.

Supported model families:

Provider	Families
OpenAI	`gpt-5`, `gpt-45`, `gpt-41`, `gpt-4o`, `o1`, `o3`, `o4`, `gpt-4`, `gpt-35`
Anthropic	`claude-3-haiku`, `claude-3-sonnet`, `claude-3-opus`, `claude-3-5-haiku`, `claude-3-5-sonnet`, `claude-3-7-sonnet`, `claude-4-opus`, `claude-4-sonnet`
Google	`gemini-1.5-flash`, `gemini-1.5-pro`, `gemini-2.0-flash`, `gemini-2.5-pro`, `gemini-2.5-flash`
Meta	`llama-3.3-8b`, `llama-3.3-70b`, `llama-4-scout`, `llama-4-maverick`
Mistral	`codestral`, `open-codestral-mamba`, `mistral`, `ministral`, `pixtral`
DeepSeek	`r1`

For full configuration reference including all `CreateArguments` options (temperature, max_tokens, reasoning_effort, etc.), see the Admin Guide.

4. Core Usage

Creating Agents

AssistantAgent -- an AI-powered agent that uses a model client to generate responses:

```python
from autogen_agentchat.agents import AssistantAgent
from autogen_ext.models.openai import OpenAIChatCompletionClient

model_client = OpenAIChatCompletionClient(model="gpt-4o")
agent = AssistantAgent(
    name="assistant",
    model_client=model_client,
    system_message="You are a helpful assistant.",
)
```

UserProxyAgent -- represents a human user in the conversation:

```python
from autogen_agentchat.agents import UserProxyAgent
```

```
user = UserProxyAgent(name="user")
```

Custom agents (Core API) -- for advanced control, extend `RoutedAgent`:

python

```
from autogen_core import RoutedAgent, message_handler, MessageContext

class MyAgent(RoutedAgent):
    def __init__(self):
        super().__init__("My custom agent")

    @message_handler
    async def handle_message(self, message: MyMessageType, ctx: MessageContext) -&gt; None:
        # Process message and optionally publish responses
        await self.publish_message(response, topic_id)
```

Building Teams (Group Chat)

Teams are the primary way to orchestrate multi-agent collaboration.

Round-Robin Group Chat -- agents take turns in order:

python

```
from autogen_agentchat.teams import RoundRobinGroupChat
from autogen_agentchat.conditions import MaxMessageTermination

team = RoundRobinGroupChat(
    participants=[agent1, agent2, agent3],
    termination_condition=MaxMessageTermination(max_messages=10),
)
result = await team.run(task="Write a short story about AI.")
```

Selector Group Chat -- an LLM selects the next speaker based on context:

python

```
from autogen_agentchat.teams import SelectorGroupChat

team = SelectorGroupChat(
    participants=[researcher, writer, reviewer],
    model_client=model_client,
    termination_condition=MaxMessageTermination(max_messages=20),
)
```

> **Warning**
>
> `SelectorGroupChat` has a known issue (#7471) where setting `allow_repeated_speaker=False` can cause a livelock when the fallback logic returns the excluded previous speaker. Always set a `MaxMessageTermination` condition as a safety net.

Termination Conditions

Control when agent conversations end:

Condition	Description

`MaxMessageTermination(max_messages=N)`	Stop after N messages
`TextMentionTermination(text="DONE")`	Stop when a message contains specific text
`StopMessageTermination()`	Stop when an agent sends a stop message

> **Note**
>
> Issue #5870 recommends using `MaxMessageTermination` or `StopMessageTermination` over `TextMentionTermination` for production use, as text-based termination can be unreliable with certain models.

Tool Use

Give agents access to tools (functions they can call):

python

```python
from autogen_core.tools import BaseTool
from pydantic import BaseModel

class CalculatorArgs(BaseModel):
    x: int
    y: int

class CalculatorResult(BaseModel):
    result: int

class CalculatorTool(BaseTool[CalculatorArgs, CalculatorResult]):
    def __init__(self):
        super().__init__(
            args_type=CalculatorArgs,
            return_type=CalculatorResult,
            name="calculator",
            description="Add two numbers",
        )

    async def run(self, args: CalculatorArgs, cancellation_token=None):
        return CalculatorResult(result=args.x + args.y)

# Assign tools to an agent
agent = AssistantAgent(
    name="math_agent",
    model_client=model_client,
    tools=[CalculatorTool()],
)
```

MCP Tool Integration

AutoGen supports the Model Context Protocol (MCP) for connecting to external tool servers:

python

```python
from autogen_ext.tools.mcp import McpWorkbench, StdioServerParams

# Connect to an MCP server
server_params = StdioServerParams(command="my-mcp-server", args=["--stdio"])
workbench = McpWorkbench(server_params=server_params)
```

```
tools = await workbench.list_tools()

agent = AssistantAgent(
    name="mcp_agent",
    model_client=model_client,
    tools=tools,
)
```

> **Tip**
>
> Use `McpSessionHost` with `ChatCompletionClientSampler` for advanced MCP features like server-initiated sampling and elicitation. See `mcp_session_host_example.py` in `autogen-ext/examples/` for a complete demo.

Streaming Responses

Stream agent responses token-by-token:

python

```
async for chunk in agent.on_messages_stream(messages, cancellation_token=None):
    if isinstance(chunk, str):
        print(chunk, end="", flush=True)
```

Console Output

Use the built-in `Console` class for formatted output:

python

```
from autogen_agentchat.ui import Console

result = await Console(team.run_stream(task="Solve this problem."))
```

5. Common Tasks

Task 1: Two-Agent Conversation

Goal: Have an AI assistant and a human proxy collaborate on a task.

python

```
import asyncio
from autogen_agentchat.agents import AssistantAgent, UserProxyAgent
from autogen_agentchat.teams import RoundRobinGroupChat
from autogen_agentchat.conditions import MaxMessageTermination
from autogen_agentchat.ui import Console
from autogen_ext.models.openai import OpenAIChatCompletionClient

async def main():
    model_client = OpenAIChatCompletionClient(model="gpt-4o")
    assistant = AssistantAgent("assistant", model_client=model_client)
    user = UserProxyAgent("user")

    team = RoundRobinGroupChat(
```

```
        participants=[user, assistant],
        termination_condition=MaxMessageTermination(max_messages=6),
    )
    result = await Console(team.run_stream(task="Help me write a Python sort function."))

asyncio.run(main())
```

Task 2: Multi-Agent Code Review Team

Goal: Build a team with a coder, a reviewer, and a tester.

python

```
coder = AssistantAgent(
    "coder",
    model_client=model_client,
    system_message="You write clean, well-documented Python code.",
)
reviewer = AssistantAgent(
    "reviewer",
    model_client=model_client,
    system_message="You review code for bugs, style, and security issues.",
)
tester = AssistantAgent(
    "tester",
    model_client=model_client,
    system_message="You write comprehensive unit tests for code.",
)

team = RoundRobinGroupChat(
    participants=[coder, reviewer, tester],
    termination_condition=MaxMessageTermination(max_messages=12),
)
result = await team.run(task="Implement a binary search function with tests.")
```

Task 3: Using AutoGen Studio

Goal: Visually build and test agent teams without writing code.

1. Launch AutoGen Studio:

bash

```
autogenstudio ui --host 127.0.0.1 --port 8081
```

2. Open `http://127.0.0.1:8081` in your browser
3. Create a new **Team** from the gallery or build custom
4. Add agents, configure model clients, and set termination conditions
5. Create a **Session** and start chatting with your team

AutoGen Studio CLI commands:

Command	Description
`autogenstudio ui`	Launch the full web UI
`autogenstudio serve --team team.json`	Serve a team as an API endpoint

`autogenstudio lite`	Launch lightweight mode for quick experimentation
`autogenstudio version`	Print the CLI version

`autogenstudio ui` **options:**

Flag	Default	Description
`--host`	`127.0.0.1`	Host address
`--port`	`8081`	Port number
`--workers`	`1`	Number of uvicorn workers
`--reload`	`false`	Auto-reload on code changes
`--docs`	`true`	Enable API documentation
`--appdir`	`~/.autogenstudio`	Application data directory
`--database-uri`	`None`	Custom database URI
`--auth-config`	`None`	Path to auth config YAML
`--upgrade-database`	`false`	Upgrade database schema

> **Tip**
>
> Use `autogenstudio lite --team team.json` for quick experimentation without the full database and gallery setup.

Task 4: Running Benchmarks with AgBench

Goal: Evaluate agent performance on standardized benchmarks.

bash

```
# Install agbench
cd autogen/python/packages/agbench
pip install -e .

# Initialize a benchmark (e.g., GAIA)
cd benchmarks/GAIA
python Scripts/init_tasks.py

# Run the benchmark
autogenbench run Tasks/gaia_validation_level_1__MagenticOne.jsonl

# View results
autogenbench tabulate Results/gaia_validation_level_1__MagenticOne/
```

AgBench commands:

Command	Description

`autogenbench run <config>`	Run a benchmark configuration
`autogenbench tabulate <results_dir>`	Tabulate results of a previous run
`autogenbench lint <logfile>`	Analyze a console log for errors
`autogenbench remove_missing <dir>`	Remove folders with missing results

Task 5: Code Execution with Docker Sandboxing

Goal: Let agents execute code safely in Docker containers.

python

```
from autogen_ext.code_executors.docker import DockerCommandLineCodeExecutor

executor = DockerCommandLineCodeExecutor(
    image="python:3.11-slim",
    timeout=60,
)

# Use with an AssistantAgent that generates code
agent = AssistantAgent(
    "coder",
    model_client=model_client,
    code_executor=executor,
)
```

> **Warning**
>
> The `LocalCommandLineCodeExecutor` executes LLM-generated code directly on your machine without sandboxing (see issue #7462). Always use `DockerCommandLineCodeExecutor` or `AzureContainerCodeExecutor` in production.

Task 6: Building a .NET Agent

Goal: Create an event-driven agent in C#.

csharp

```
using Microsoft.AutoGen.Core;
using Microsoft.AutoGen.Contracts;

var builder = new AgentsAppBuilder();
builder.UseInProcessRuntime();
builder.AddAgent&lt;HelloAgent&gt;("hello");
var app = await builder.BuildAsync();
await app.StartAsync();

// Publish a message
await app.AgentRuntime.PublishMessageAsync(
    new TextMessage { Content = "Hello!" },
    new TopicId("hello", "default")
);

await app.WaitForShutdownAsync();
```

Task 7: Human-in-the-Loop Conversations

Goal: Build an agent that pauses for human approval using an intervention handler.

python

```
from autogen_core import DefaultInterventionHandler

class ApprovalHandler(DefaultInterventionHandler):
    async def on_publish(self, message, ctx):
        if needs_approval(message):
            approval = input("Approve? (y/n): ")
            if approval != "y":
                return None  # Block the message
        return message
```

Task 8: Distributed Group Chat via gRPC

Goal: Run agents across multiple processes using gRPC.

python

```
from autogen_ext.runtimes.grpc import GrpcWorkerAgentRuntime

runtime = GrpcWorkerAgentRuntime(host_address="localhost:50051")
await runtime.start()

# Register agents on this worker
await runtime.register_agent_type("writer", agent_factory)
```

> **Note**
>
> Distributed runtimes require a gRPC gateway service. The .NET `AgentHost` Dockerfile exposes port 5001 for this purpose.

Task 9: Using Reasoning Models

Goal: Leverage reasoning models (o1, o3) with adjustable effort.

python

```
client = OpenAIChatCompletionClient(
    model="o3",
    reasoning_effort="medium",  # minimal, low, medium, high
)
```

Task 10: Caching Model Responses

Goal: Cache LLM responses to reduce costs during development.

python

```
from autogen_ext.models.cache import ChatCompletionCache
from autogen_ext.cache_store import DiskCacheStore

cache_store = DiskCacheStore(cache_dir=".cache/autogen")
cached_client = ChatCompletionCache(client=model_client, cache_store=cache_store)
```

Task 11: GraphRAG-Augmented Agents

Goal: Combine agents with knowledge graphs for enhanced retrieval.

bash

```
cd autogen/python/samples/agentchat_graphrag
pip install -r requirements.txt
python app.py --verbose
```

The sample downloads knowledge graph data and creates an agent team that queries it through the GraphRAG tool.

Task 12: Serving Teams as FastAPI Endpoints

Goal: Expose an agent team as an HTTP API.

bash

```
autogenstudio serve --team team.json --host 0.0.0.0 --port 8084
```

This starts a FastAPI server that accepts chat requests and routes them to the team defined in the JSON file.

6. Troubleshooting

Diagnostic Decision Tree

text

```
Problem with AutoGen?
|-- Import error? -&gt; Check package installation (Section 2)
|-- API key error? -&gt; Verify environment variables (Section 3)
|-- Agent not responding? -&gt; Check model client configuration
|-- Group chat loops forever? -&gt; Add MaxMessageTermination condition
|-- Code execution fails? -&gt; Verify Docker is running
+-- AutoGen Studio won't start? -&gt; Check port availability and dependencies
```

Error: `ModuleNotFoundError: No module named 'autogen_agentchat'`

Cause: The `autogen-agentchat` package is not installed, or you're in the wrong virtual environment.

Solution:

1. Activate your virtual environment
2. Install the package:

bash

```
pip install autogen-agentchat
```

Prevention: Use a `requirements.txt` or `pyproject.toml` to track dependencies.

Error: `Missing required field 'family' in ModelInfo`

Cause: Starting in v0.4.7, the `family` field is required in `ModelInfo`. Custom model clients must provide it.

Solution:

```python
model_info = {
    "vision": True,
    "function_calling": True,
    "json_output": True,
    "family": "gpt-4o",
    "structured_output": True,
}
```

Prevention: Always specify the `family` field when creating custom model client configurations.

Error: `SelectorGroupChat livelock` (#7471)

Cause: When `allow_repeated_speaker=False`, the fallback logic may return the excluded previous speaker, causing an infinite loop.

Solution:

1. Always set a `MaxMessageTermination` condition as a safety net
2. Monitor agent conversation length
3. Consider using `RoundRobinGroupChat` if speaker selection is not critical

Prevention: Pair `SelectorGroupChat` with a `MaxMessageTermination` condition.

Error: `ValueError: Unknown tool: {call.name}`

Cause: An agent attempted to call a tool that is not registered with its runtime.

Solution:

1. Verify all tools are passed to the agent's `tools` parameter
2. Ensure tool names match between the model's function call and the registered tool

Prevention: Log tool registrations at startup and verify against expected names.

Error: `LocalCommandLineCodeExecutor` security warning (#7462)

Cause: LLM-generated code runs unsandboxed on the host machine.

Solution: Switch to a sandboxed executor:

```python
from autogen_ext.code_executors.docker import DockerCommandLineCodeExecutor
executor = DockerCommandLineCodeExecutor(image="python:3.11-slim")
```

> **Caution**
>
> Never use `LocalCommandLineCodeExecutor` in production or with untrusted inputs. LLM-generated code can execute arbitrary system commands.

Error: AutoGen Studio `Auth config file not found`

Cause: The `--auth-config` flag points to a nonexistent file.

Solution:

1. Verify the path: `ls /path/to/auth_config.yaml`
2. If not using auth, omit the `--auth-config` flag entirely

Prevention: Use absolute paths for the auth config file.

Error: gRPC protobuf import errors

Cause: Generated protobuf files have incorrect relative import paths.

Solution: Run the import fixup script from the Python directory:

bash

```
cd autogen/python
python fixup_generated_files.py
```

Prevention: Re-run the fixup script after regenerating protobuf files.

Error: `Image content cannot be converted to text`

Cause: Attempting to pass image content to a component that only supports text (e.g., `Teachability` memory).

Solution: Ensure image content is only sent to model clients with `vision: true` in their `ModelInfo`.

> **Warning**
> Not all model providers support multimodal input. Check `model_info["vision"]` before sending images.

Error: `Invalid role selected` in distributed group chat

Cause: The `GroupChatManager` LLM selector returned a participant name that doesn't match any registered topic type.

Solution:

1. Ensure participant descriptions are clear and unambiguous
2. Verify topic types match between agent registrations and the manager's participant list

Error: `Config file must contain a JSON/YAML object`

Cause: A model configuration file is empty, contains an array, or is malformed.

Solution: Ensure the config file is a valid JSON or YAML object:

yaml

```
model: gpt-4o
api_key: sk-...
```

Error: `Unsupported server params type` in MCP

Cause: An MCP connection was attempted with an unrecognized server parameter type.

Solution: Use one of the supported parameter types: `StdioServerParams`, `SseServerParams`, or `StreamableHttpServerParams`.

7. FAQ

Q: What is the difference between `autogen-core`, `autogen-agentchat`, and `autogen-ext`?

`autogen-core` provides the foundational runtime, message types, and agent base classes. `autogen-agentchat` builds on core to provide high-level conversational agents (`AssistantAgent`, `UserProxyAgent`) and team patterns (`RoundRobinGroupChat`, `SelectorGroupChat`). `autogen-ext` provides extensions for model clients, code executors, tools, and memory stores.

Q: How does AutoGen v0.4 differ from the previous v0.2 API?

AutoGen v0.4+ is a complete rewrite with an event-driven architecture, async-first design, and modular package structure. The `pyautogen` package provides backwards compatibility. New projects should use `autogen-agentchat` and `autogen-core` directly.

Q: Can I use AutoGen with local models (no API key)?

Yes. Use `OllamaChatCompletionClient` for Ollama-served models or `LlamaCppChatCompletionClient` for llama.cpp models. The `LMStudioAgent` (.NET) connects to LM Studio's local OpenAI-compatible API.

Q: Can Python and .NET agents communicate?

Yes. AutoGen supports cross-language agent communication through gRPC and the CloudEvents messaging protocol. Python agents use `GrpcWorkerAgentRuntime` and .NET agents use the `AgentHost` runtime gateway to connect to a shared gRPC channel.

Q: What is Magentic-One?

Magentic-One is a multi-agent system built on AutoGen for complex web-based tasks. It includes specialized agents (WebSurfer, FileSurfer, Coder, ComputerTerminal, Orchestrator) that collaborate to complete tasks requiring web browsing and code execution. Install with `pip install autogen-ext[openai,magentic-one]`.

Q: How do I use AutoGen with Streamlit?

The `python/samples/agentchat_streamlit` sample shows how to build a Streamlit chat interface backed by an AutoGen agent. Create an `Agent` class that wraps a `ChatCompletionClient` and call its `chat()` method from Streamlit's chat input handler.

Q: What authentication providers does AutoGen Studio support?

AutoGen Studio supports GitHub OAuth, Microsoft MSAL, Firebase authentication, and a no-auth mode for local development. Configure via a YAML file passed to `--auth-config`.

Q: How do I handle rate limits with model providers?

Use the `ChatCompletionCache` with a `DiskCacheStore` or `RedisCacheStore` to cache responses during development. For production, configure `max_retries` in the model client configuration.

Q: What benchmarks are available?

AgBench includes GAIA (general AI assistant tasks) and HumanEval (code generation) benchmarks. Both support configurable agent templates and automated result tabulation.

Q: Is there a way to persist agent state?

In .NET, use `Store()` and `Read<AgentState>()` for agent state persistence. In Python, custom agents can implement `save_state()` and `load_state()` methods. The `Teachability` memory extension provides persistent learning across conversations.

Q: How do I contribute to AutoGen?

Clone the repository, install development dependencies with `uv sync`, and run the test suite with `poe mypy` and `poe pyright`. Documentation is built with Sphinx: `poe docs-build`. The code uses `ruff` for formatting and linting.

Q: Can I use Anthropic models with extended thinking?

Yes. Configure the `AnthropicChatCompletionClient` with a `thinking` parameter. The response's `thought` field will contain the model's reasoning text. This works with both streaming and non-streaming calls.

> ***DocCompiler.ai*** *v3.1 | Upgraded source coverage | Generated 2026-05-13 | Source:* `microsoft/autogen` *@* `027ecf0` *A Cyber Panda Solutions LLC product. This document is intentionally comprehensive -- built to be the complete, source-verified reference an AI or human needs without consulting other sources.* ***Do not print:*** *the always-current version lives at doccompiler.ai/microsoft/autogen.*