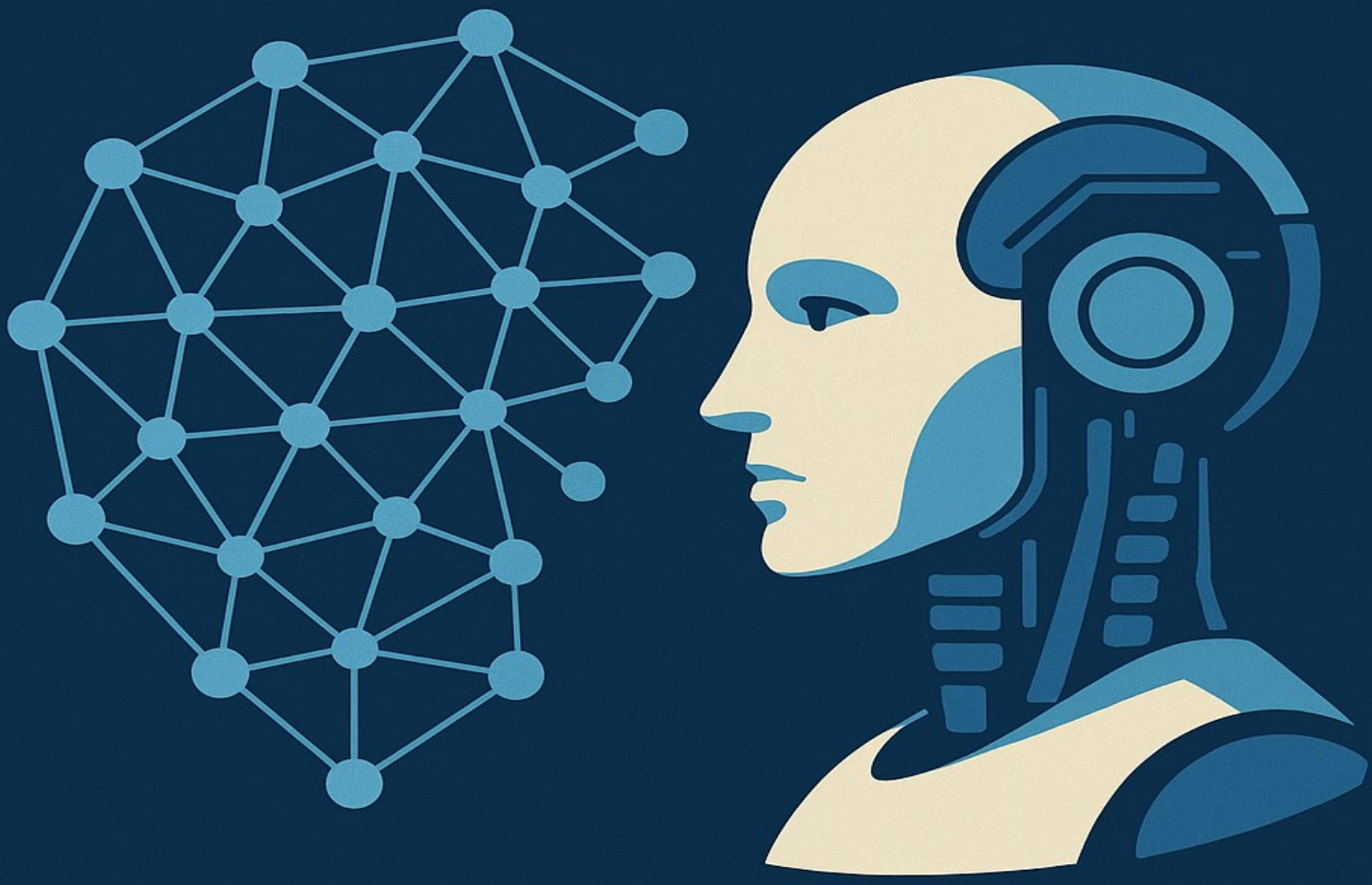
Build AutoGen Agents with Qwen3

Structured Output & Thinking Mode



AI Books

2025

Build AutoGen Agents with Qwen3: Structured Output & Thinking Mode

This book will walk you through integrating AutoGen with Qwen3, including how to enable structured output for Qwen3 in AutoGen and manage Qwen3's thinking mode capabilities.
If you're in a hurry for the solution, you can skip the "how-to" sections and jump straight to the end, where I've shared all the source code. Feel free to use and modify it without asking for permission.

Introduction

As enterprises begin deploying Qwen3 models, corresponding agent frameworks must adapt to fully utilize Qwen3's capabilities.
For the past two months, my team and I have been working on a large-scale project using AutoGen. Like LlamaIndex Workflow, this event-driven agent framework allows our agents to integrate better with enterprise message pipelines, leveraging the full power of our data processing architecture.
If you're also interested in LlamaIndex Workflow

We've spent considerable time with Qwen3, experimenting with various approaches and even consulting directly with the Qwen team on certain options.
I'm confident this article will help you, even if you're not using Qwen series models or AutoGen specifically. The problem-solving approaches are universal, so you'll save significant time.

Why should I care?

There's an old Chinese saying: "A craftsman must first sharpen his tools." To fully benefit from the latest technology's performance improvements and development conveniences, integrating models into existing systems is the first step.

This article will cover:

- Creating an OpenAI-like client that lets AutoGen connect to Qwen3 via OpenAI API.
- Exploring AutoGen's structured output implementation and alternative approaches, ultimately adding structured output support for Qwen3.
- Supporting Qwen3's `extra_body` parameters in AutoGen by controlling the thinking mode toggle.
- Finally, we'll put these lessons into practice with an article-summarizing agent project.

Let's begin!

Step 1: Building an OpenAILike Client

Testing the official OpenAI client

We know both Qwen and DeepSeek models support OpenAI API calls. AutoGen provides a `OpenAIChatCompletionClient` class for GPT series models. Can we use it to connect to public cloud or privately deployed Qwen3 models? Unfortunately not. When we tried:

```
original_model_client = OpenAIChatCompletionClient(
    model="qwen-plus-latest",
    base_url=os.getenv("OPENAI_BASE_URL")
)

agent = AssistantAgent(
    name="assistant",
    model_client=original_model_client,
    system_message="You are a helpful assistant."
)
```

We encountered an error:

```
  File "D:\Documents\PythonProject\agentic-ai-playground\venv\Lib\site-packages\autogen_ext\models\openai\_model_info.py",
    raise ValueError("model_info is required when model name is not a valid OpenAI model")
ValueError: model_info is required when model name is not a valid OpenAI model
```

When using OpenAIChatCompletionClient, you need to specify an OpenAI model.

Image by Author

Checking the `_model_client.py` file reveals that `OpenAIChatCompletionClient` only supports GPT series models, plus some Gemini and Claude models - no promises for others.

But this isn't new. Remember when the OpenAI client last restricted model types? Exactly — LlamaIndex's OpenAI client had similar limitations, but the community provided an OpenAILike client as a workaround.

How to Connect LlamaIndex with Private LLM API Deployments

When your enterprise doesn't use public models like OpenAI

medium.com

Our solution here is similar: we'll build an OpenAILike client supporting Qwen (and DeepSeek) series models.

Trying the model_info parameter

Checking the API docs reveals a `mode_info` parameter: *"Required if the model name is not a valid OpenAI model."*

So `OpenAIChatCompletionClient` can support non-OpenAI models if we provide the model's own `mode_info`.

For Qwen3 models on public cloud, `qwen-plus-lastest` and `qwen-turbo-latest` are the newest. I'll demonstrate with qwen-plus-latest:

```
original_model_client = OpenAIChatCompletionClient(
    model="qwen-plus-latest",
    base_url=os.getenv("OPENAI_BASE_URL"),
    model_info={
        "vision": False,
        "function_calling": True,
        "json_output": True,
        "family": 'qwen',
```

```
            "structured_output": True,
            "multiple_system_messages": False,
        }
    )

    ...

    async def main():
        await Console(
            agent.run_stream(task="Hi, could you introduce yourself? Are you Qwen3?")
        )
```

```
---------- TextMessage (user) ----------
Hi, could you introduce yourself? Are you Qwen3?
---------- TextMessage (assistant) ----------
Hello! I am Qwen3, the latest large language model launched by Alibaba Group. Based on broader knowledge sources, I offe
r enhanced language understanding and expression capabilities. Compared with previous versions, I have significant impro
vements in several aspects:

1. **Stronger and More Comprehensive Foundation Model**: Based on broader knowledge sources, I offer enhanced language u
nderstanding and expression capabilities, available in multiple versions—including dense and Mixture-of-Experts (MoE)—to
 suit diverse application scenarios.

2. **Diverse Selection of Reasoning Modes**: I support switching between various reasoning modes, including a chat mode
ideal for everyday communication and a reasoning mode optimized for deep thinking, effectively addressing users' complex
 and precise interaction requirements.

3. **Exceptional Reasoning Capabilities**: My logical reasoning, mathematical calculations, and code generation abilitie
s have been significantly enhanced and optimized, enabling efficient and accurate execution of challenging reasoning tas
ks.

4. **Comprehensive Conversational Skills**: Through iterative improvements and deep optimization, I can fluently perform
 content creation, multi-turn dialogues, role-playing, and even support intelligent agent interactions, delivering a mor
e natural and immersive conversational experience.

5. **Expanded Multilingual Support**: My linguistic capabilities now span over 100 major global languages, accommodating
 diverse international communication needs and effectively serving users worldwide.
```

After adding custom model_info, the client successfully connected to the Qwen3 model. Image by Author

The model connects successfully and generates content normally. But this creates new headaches — I don't want to copy-paste `model_info` constantly, nor do I care about its various options.

Solution? Time to implement our own OpenAILike that encapsulates this information.

Implementing OpenAILikeChatCompletionClient

We'll implement this through inheritance. The code lives in `utils/openai_like.py`.

By inheriting from `OpenAIChatCompletionClient`, we'll automate `model_info` handling.

First, we compile all potential models' `model_info` into a dict, with a default `model_info` for unlisted models.

```
_MODEL_INFO: dict[str, dict] = {
    ...
    "qwen-plus-latest": {
        "vision": False,
        "function_calling": True,
        "json_output": True,
        "family": ModelFamily.QWEN,
        "structured_output": True,
        "context_window": 128_000,
        "multiple_system_messages": False,
    },
    ...
}

DEFAULT_MODEL_INFO = {
    "vision": False,
    "function_calling": True,
    "json_output": True,
    "family": ModelFamily.QWEN,
    "structured_output": True,
    "context_window": 32_000,
    "multiple_system_messages": False,
}
```

In `__init__`, we check if users provided `model_info`. If not, we look up the model parameter in our config, falling back to default if missing.
Since `OpenAIChatCompletionClient` requires users to provide `base_url`, we've optimized this too: if missing, we'll pull from `OPENAI_BASE_URL` or `OPENAI_API_BASE` environment variables.
Our final `__init__` method looks like:

```
class OpenAILikeChatCompletionClient(OpenAIChatCompletionClient):
    def __init__(self, **kwargs):
        self.model = kwargs.get("model", "qwen-max")
        if "model_info" not in kwargs:
            kwargs["model_info"] = _MODEL_INFO.get(self.model, DEFAULT_MODEL_INFO)
        if "base_url" not in kwargs:
            kwargs["base_url"] = os.getenv("OPENAI_BASE_URL") or
os.getenv("OPENAI_API_BASE")

        super().__init__(**kwargs)
```

Let's test our `OpenAILikeChatCompletionClient`:

```
model_client = OpenAILikeChatCompletionClient(
    model="qwen-plus-latest"
)

agent = AssistantAgent(
    name="assistant",
    model_client=model_client,
    system_message="You are a helpful assistant."
)
```

Done! Just specify the model and we're ready to use the latest Qwen3.

Step 2: Supporting structured_output

Structured_output specifies a pydantic BaseModel-derived class as standard output. This provides consistent, predictable output formats for more precise agent messaging.

For enterprise applications using frameworks and models, structured_output is essential.

AutoGen's structured_output implementation

AutoGen supports structured_output — just implement a pydantic `BaseModel` class and pass it via `output_content_type` to AssistantAgent. The agent's response then becomes a `StructuredMessage` containing structured output.

Let's test Qwen3's structured_output capability.

Following official examples, we'll create a sentiment analysis agent. First, define a data class:

```
class AgentResponse(BaseModel):
    thoughts: str
    response: Literal["happy", "sad", "neutral"]
```

Then pass this class to the agent via `output_content_type`:

```
structured_output_agent = AssistantAgent(
    name="structured_output_agent",
    model_client=model_client,
```

```
    system_message="Categorize the input as happy, sad, or neutral following json
format.",
    output_content_type=AgentResponse
)
```

Running this agent produces an error because the model's JSON output doesn't match our class definition, suggesting the model didn't receive our parameters:

```
pydantic_core._pydantic_core.ValidationError: 2 validation errors for AgentResponse
thoughts
  Field required [type=missing, input_value={'sentiment': 'happy'}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.11/v/missing
response
  Field required [type=missing, input_value={'sentiment': 'happy'}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.11/v/missing
```

The model's JSON output doesn't match our class definition. Image by Author

Why? Does Qwen not support structured_output? To answer, we need to understand AutoGen's structured_output implementation.

When working directly with LLMs, structured_output typically adjusts the chat completion API's `response_format` parameter.

Having modified `OpenAIChatCompletionClient` earlier, we check its code for structured_output references and find this comment:

```
# Structured output response, with a pre-defined JSON schema.
{
    "type": "json_schema",
    "json_schema": {
        "name": "name of the schema, must be an identifier.",
        "description": "description for the model.",
        # You can convert a Pydantic (v2) model to JSON schema
        # using the `model_json_schema()` method.
        "schema": "<the JSON schema itself>",
        # Whether to enable strict schema adherence when
        # generating the output. If set to true, the model will
        # always follow the exact schema defined in the
        # `schema` field. Only a subset of JSON Schema is
        # supported when `strict` is `true`.
        # To learn more, read
        # https://platform.openai.com/docs/guides/structured-outputs.
        "strict": False,  # or True
    },
}
```

Comment of structured_output in OpenAIChatCompletionClient. Image by Author

This suggests that with structured_output, OpenAI client's `response_format` parameter is set to:

```
{
    "type": "json_schema",
    "json_schema": {
        "name": "name of the schema, must be an identifier.",
        "description": "description for the model.",
        "schema": "<the JSON schema itself>",
        "strict": False,  # or True
    },
}
```

But Qwen3's documentation shows its response_format only supports `{"type": "text"}` and `{"type": "json_object"}`, not `{"type": "json_schema"}`.

Does this mean Qwen3 can't do structured_output?

Not necessarily. The essence of structured_output is getting the model to output JSON matching our schema. Without `response_format`, we have other solutions.

Implementing structured_output via function calling

Returning to Python's nature: in Python, all classes are callable objects like functions, including pydantic `BaseModel` classes.
Can we leverage this with LLM function calling for structured_output?
Absolutely — by treating data classes as special functions.
Let's modify our agent. Instead of `output_content_type`, we'll use tools parameter, passing `AgentResponse` as a tool. The model will then call this tool for output:

```
function_calling_agent = AssistantAgent(
    name="function_calling_agent",
    model_client=model_client,
    system_message="Categorize the input as happy, sad, or neutral following json
format.",
    tools=[AgentResponse],
)
```

Results:

```
---------- TextMessage (user) ----------
I'm happy.
---------- ToolCallRequestEvent (function_calling_agent) ----------
[FunctionCall(id='call_1092f89b7e2747648ddeeb', arguments='{"response": "happy",
ss."}', name='AgentResponse')]
---------- ToolCallExecutionEvent (function_calling_agent) ----------
[FunctionExecutionResult(content='{"thoughts": "The user expressed happiness.", "
se', call_id='call_1092f89b7e2747648ddeeb', is_error=False)]
---------- ToolCallSummaryMessage (function_calling_agent) ----------
{"thoughts": "The user expressed happiness.", "response": "happy"}
```

The model outputs JSON perfectly when using function calling. Image by Author

The model outputs JSON matching our class's schema perfectly. This works!
However, with multiple tools, the agent sometimes ignores the data class tool, outputting freely.
Is there a more stable approach? Let's think deeper: structured_output's essence is getting JSON matching our schema. Can we leverage that directly?

Making the model output according to json_schema

Having the model output according to `json_schema` is entirely feasible. AutoGen previously used response_format's `json_schema`, but we can also specify the schema directly in `system_prompt`:

```
json_schema_agent = AssistantAgent(
    name="json_schema_agent",
    model_client=model_client,
    system_message=dedent(f"""
    Categorize the input as happy, sad, or neutral,
    And follow the JSON format defined by the following JSON schema:
    {AgentResponse.model_json_schema()}
    """)
)
```

Results:

```
---------- TextMessage (user) ----------
I'm happy.
---------- TextMessage (json_schema_agent) ----------
{
  "thoughts": "The user expressed a positive emotion.",
  "response": "happy"
}
```

When we directly specify json_schema in the system_prompt, the model outputs perfect JSON content. Image by Author

The agent outputs JSON matching our schema. Understanding the principles makes structured_output implementation straightforward.

We can further convert JSON output back to our data class for code processing:

```
result = await Console(json_schema_agent.run_stream(task="I'm happy."))
structured_result = AgentResponse.model_validate_json(
    result.messages[-1].content
)
print(structured_result.thoughts)
print(structured_result.response)
```

```
---------- TextMessage (user) ----------
I'm happy.
---------- TextMessage (json_schema_agent) ----------
{
  "thoughts": "The user expressed a positive emotion.",
  "response": "happy"
}
The user expressed a positive emotion.
happy
```

We can manually convert JSON text into Pydantic data classes. Image by Author

Perfect.

But specifying `json_schema` in `system_prompt` is cumbersome. Can we make Qwen3 agents support `output_content_type` directly?

Making Qwen3 support AutoGen's output_content_type parameter

Our ultimate goal is framework-level Qwen3 support for structured_output via `output_content_type`, without changing AutoGen usage.

Earlier we saw `output_content_type` works via `response_format`, set when agents call OpenAI Client's `create` or `create_stream` methods.

We also know modifying `system_prompt` to explicitly specify `json_schema` produces stable structured output.

Having implemented `OpenAILikeChatCompletionClient`, can we override `create` and `create_stream` to modify `system_prompt`?

Let's do it. First, `add _append_json_schema` to `OpenAILikeChatCompletionClient`. This finds the first message in the sequence and appends `json_schema` instructions to `system_prompt`:

```
class OpenAILikeChatCompletionClient(OpenAIChatCompletionClient):
    ...

    def _append_json_schema(self, messages: Sequence[LLMMessage],
                            json_output: BaseModel) -> Sequence[LLMMessage]:
        messages = copy.deepcopy(messages)
        first_message = messages[0]
        if isinstance(first_message, SystemMessage):
            first_message.content += dedent(f"""\

            <output-format>
            Your output must adhere to the following JSON schema format,
            without any Markdown syntax, and without any preface or explanation:
```

```
            {json_output.model_json_schema()}
            </output-format>
            """)
        return messages
```

Then override `create` and `create_stream` to call `_append_json_schema` first, while clearing `json_output` to prevent AutoGen from setting `response_format`:

```
class OpenAILikeChatCompletionClient(OpenAIChatCompletionClient):
    ...

    @override
    async def create(
            self,
            messages: Sequence[LLMMessage],
            *,
            tools: Sequence[Tool | ToolSchema] = [],
            json_output: Optional[bool | type[BaseModel]] = None,
            extra_create_args: Mapping[str, Any] = {},
            cancellation_token: Optional[CancellationToken] = None,
    ) -> CreateResult:
        if json_output is not None and issubclass(json_output, BaseModel):
            messages = self._append_json_schema(messages, json_output)
            json_output = None
        result = await super().create(
            messages=messages,
            tools=tools,
            json_output=json_output,
            extra_create_args=extra_create_args,
            cancellation_token=cancellation_token
        )
        return result
```

Our `OpenAILikeChatCompletionClient` modifications are complete. Since we modified underlying methods, users' `output_content_type` usage remains unchanged.
Let's test with a new agent, setting neither tools nor requiring `system_prompt` modifications - just `output_content_type` as per AutoGen docs:

```
structured_output_agent = AssistantAgent(
    name="structured_output_agent",
    model_client=model_client,
```

```
    system_message="Categorize the input as happy, sad, or neutral following json
format.",
    output_content_type=AgentResponse
)
```

Now the agent outputs correct JSON directly:

```
---------- TextMessage (user) ----------
I'm happy
---------- StructuredMessage[AgentResponse] (structured_output_agent) ----------
{"thoughts":"The user expressed a positive emotion.","response":"happy"}
True
```

AutoGen generated StructuredMessage this time. Image by Author

Notice the message type is `StructuredMessage` - the agent directly produced our data class, confirmed by isinstance checks:

```
result = await Console(
    structured_output_agent.run_stream(task="I'm happy")
)
print(isinstance(result.messages[-1].content, AgentResponse))
```

With these changes, AutoGen can correctly generate structured messages in multi-agent systems using Qwen3. These modifications also work for older Qwen models and DeepSeek series.

Step 3: Supporting Thinking Mode

Parameters for enabling/disabling thinking mode

In previous examples, we used public cloud Qwen3 models (qwen-plus-latest), intentionally ignoring Qwen3's new thinking capability.

Enterprise applications typically use privately deployed `qwen3-235b-a22b` or `qwen3-30b-a3b` models.

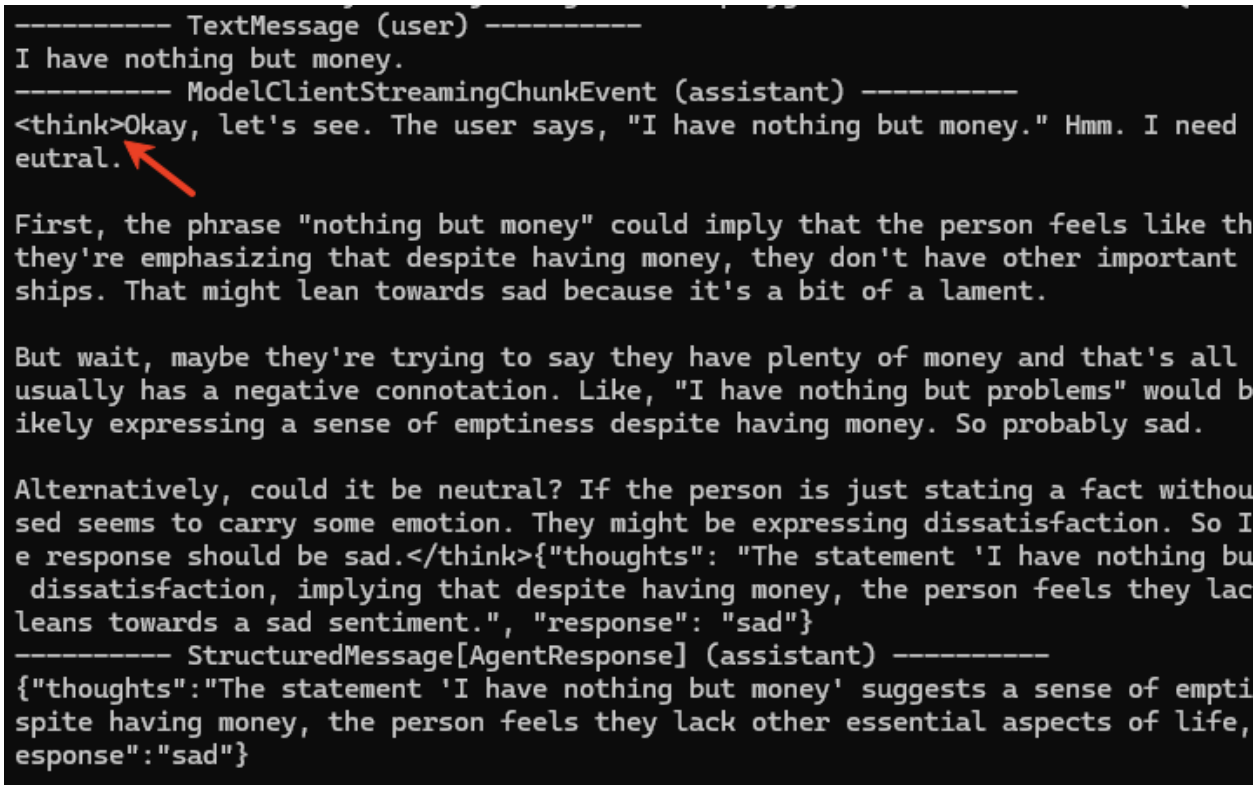
These open-source models differ by defaulting to thinking mode. Before answering, the model performs Chain of Thoughts (CoT) reasoning, significantly improving performance — similar to DeepSeek-R1 or QwQ models.

But in multi-model applications, we sometimes want to disable thinking to reduce token usage and latency.

Qwen3's documentation shows the `extra_body={"enable_thinking": xxx}` parameter controls thinking mode.
Let's test this during client creation:

```
model_client = OpenAILikeChatCompletionClient(
    model="qwen3-30b-a3b",
    extra_body={"enable_thinking": False}
)

...

async def main():
    await Console(
        agent.run_stream(task="I have nothing but money.")
    )
```

Surprisingly, this parameter has no effect:



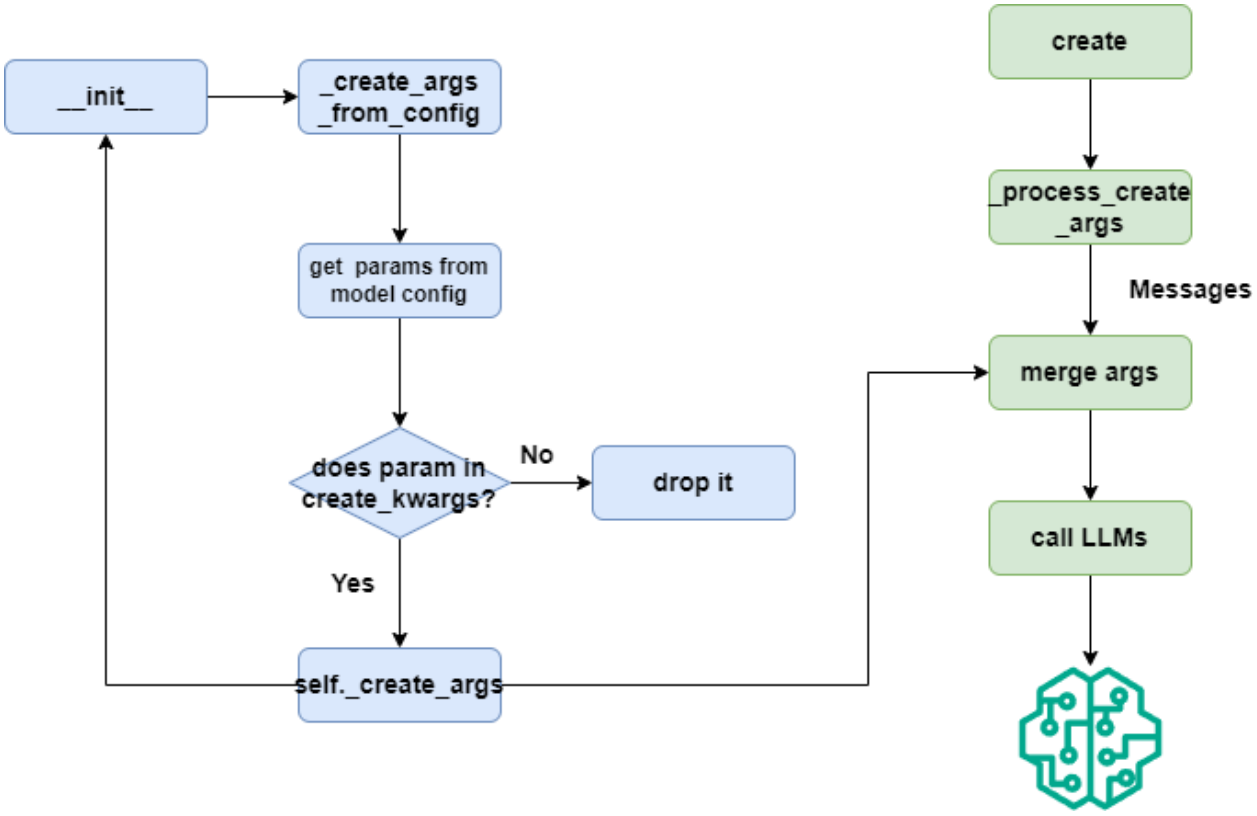
Adding the extra_body parameter directly won't have any effect. Image by Author

Why? Again, we examine AutoGen's source code.

How AutoGen handles parameters

In `OpenAIChatCompletionClient`, `__init__` calls `_create_args_from_config` to initialize parameters stored in `self._create_args`.

Then `_process_create_args` merges `self._create_args` with create's parameters before sending to the model.



How AutoGen handles parameters. Image by Author

So we can override `_process_create_args` in `OpenAILikeChatCompletionClient` to see what parameters AutoGen sends to Qwen3. Here we'll just examine `self._create_args`:

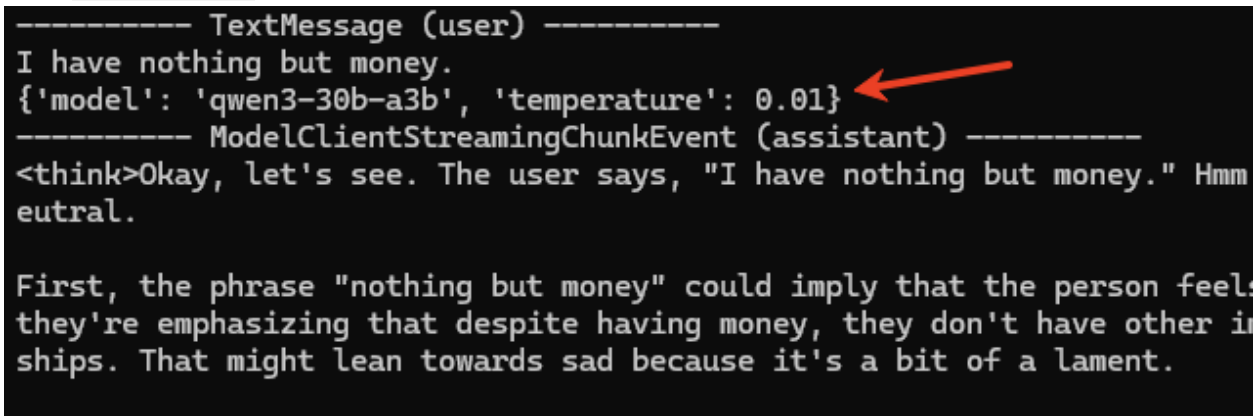
```
class OpenAILikeChatCompletionClient(OpenAIChatCompletionClient):
    ...

    def _process_create_args(
            self,
            messages: Sequence[LLMMessage],
            tools: Sequence[Tool | ToolSchema],
            json_output: Optional[bool | type[BaseModel]],
            extra_create_args: Mapping[str, Any],
    ) -> CreateParams:
        print(self._create_args)
        params = super()._process_create_args(
            messages=messages,
            tools=tools,
            json_output=json_output,
            extra_create_args=extra_create_args
```

```
        )
        return params
```

For comparison, we'll add a `temperature` parameter (which GPT models support):

```
model_client = OpenAILikeChatCompletionClient(
    model="qwen3-30b-a3b",
    temperature=0.01,
    extra_body={"enable_thinking": False}
)
```

The output shows Qwen3 receives `model` and `temperature` parameters, but not `extra_body`.



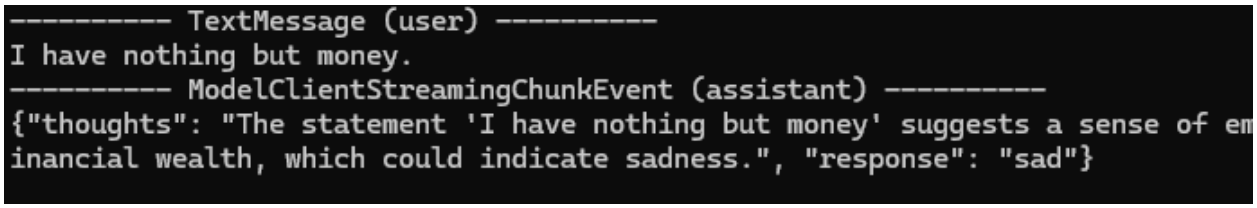
The output shows Qwen3 receives `model` and `temperature` parameters, but not `extra_body`.

Image by Author

`_create_args_from_config` checks if parameters are GPT-supported, ignoring others. Since `extra_body` is Qwen3-specific, it's ignored.

Adding extra_body support

Don't worry — we'll add `extra_body` to `self._create_args`. First, let's confirm thinking mode disabled:



The LLM no longer goes through the thinking process before generating the final result. Image by Author

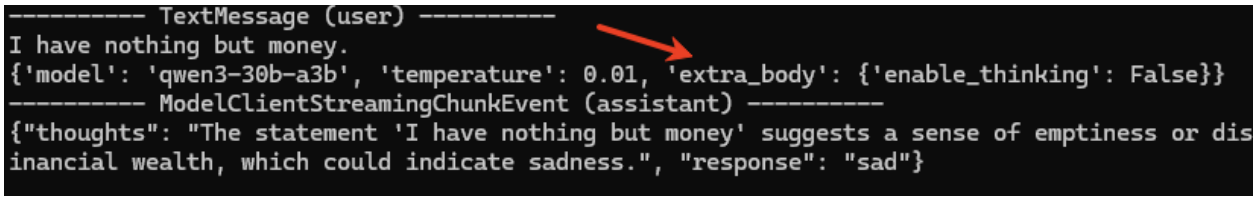
Supporting `extra_body` is simple. Knowing how `self._create_args` is generated, we just add `extra_body` after parent class initialization. `_create_args_from_config` handles constructor parameters, but as a standalone function, it's not easily overridden. Instead, in `OpenAILikeChatCompletionClient.__init__`, we'll add our parameters after `self._create_args` is created:

```
class OpenAILikeChatCompletionClient(OpenAIChatCompletionClient):
    def __init__(self, **kwargs):
        self.model = kwargs.get("model", "qwen-max")
        if "model_info" not in kwargs:
            kwargs["model_info"] = _MODEL_INFO.get(self.model, DEFAULT_MODEL_INFO)
        if "base_url" not in kwargs:
            kwargs["base_url"] = os.getenv("OPENAI_BASE_URL") or
os.getenv("OPENAI_API_BASE")

        super().__init__(**kwargs)
        for key in extra_kwargs: # Add the model-specific extension parameters for
Qwen3 in self._create_args
            if key in kwargs:
                self._create_args[key] = kwargs[key]
```

Now let's test by adding `extra_body={"enable_thinking": False}` during `model_client` creation:

```
model_client = OpenAILikeChatCompletionClient(
    model="qwen3-30b-a3b",
    temperature=0.01,
    extra_body={"enable_thinking": False}
)
```

Checking `self._create_args` again, `extra_body` is successfully included:



`extra_body` is successfully included. Image by Author

Alternative thinking mode control methods

Beyond `extra_body`, official documentation suggests two other thinking mode controls:

First: append `/think` or `/no_think` to user input for agent-level control.

```
messages = [
    {"role": "user", "content": "Give me a short introduction to large language
models./no_think"},
]
messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]
# print(messages[-1]["content"])

messages.append({"role": "user", "content": "In a single sentence./think"})
messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]
# print(messages[-1]["content"])
```

Testing shows only `/no_think` works; `/think` doesn't.
Second: append assistant-role message "`<think>\n\n</think>\n\n`" after each user input to temporarily disable thinking mode.

```
messages = [
    {"role": "user", "content": "Give me a short introduction to large language
models."},
    {"role": "assistant", "content": "<think>\n\n</think>\n\n"},
]
messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]
# print(messages[-1]["content"])

messages.append({"role": "user", "content": "In a single sentence."})
messages = generator(messages, max_new_tokens=32768)[0]["generated_text"]
# print(messages[-1]["content"])
```

Testing shows this doesn't work.
Thus, the most reliable method remains adding `extra_body` during `model_client` initialization.

Other extra_body options

`extra_body` controls other Qwen3 features too:

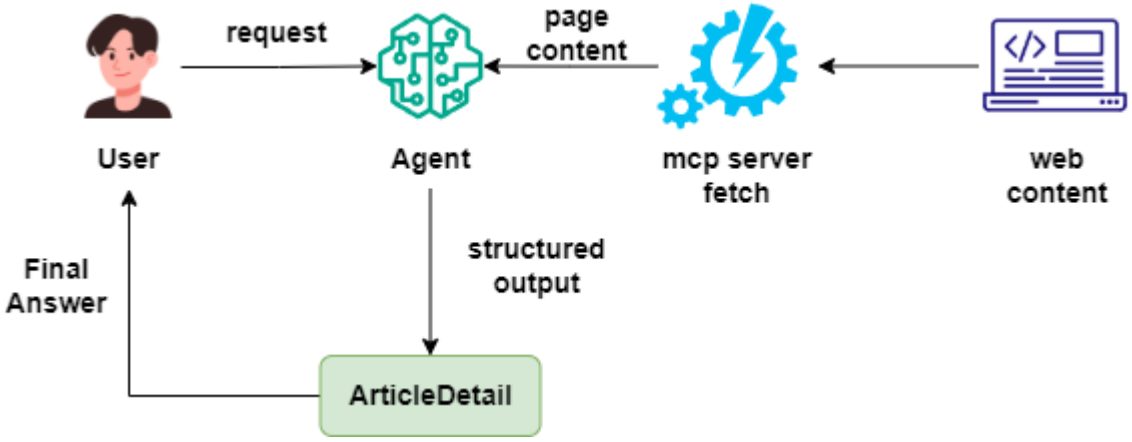
top_k: Controls sampling candidate set size during generation. Configure via extra_body={"top_k":xxx}.

thinking_budget: Maximum thinking length, only effective when `enable_thinking=True`. Configure via `extra_body={"thinking_budget": xxx}`

translation_options: For translation models, configures source/target languages, e.g., `extra_body={"translation_options": { "source_lang": "auto", "target_lang": "English" }}`. Use "auto" for mixed languages.
enable_search: Whether to reference web searches before generation. Configure via `extra_body={"enable_search": True}`

Practice Exercise: Article-Summarizing Agent

Having learned to connect AutoGen with Qwen3, support structured_output, and control thinking mode, let's test our knowledge with a small exercise. We'll use Qwen3 and mcp fetch server to create an agent that summarizes online articles with structured output.



The business flow diagram of this agent practice. Image by Author

First, initialize a `model_client` using `qwen3-30b-a3b` with thinking mode disabled:

```
model_client = OpenAILikeChatCompletionClient(
    model='qwen3-30b-a3b',
    temperature=0.01,
    extra_body={"enable_thinking": False}
)
```

For structured output, we'll define a data class extracting `title`, `url`, `author`, `keywords`, and `summary` from articles, adding descriptions for clarity:

```
class ArticleDetail(BaseModel):
    title: str
    url: str
    author: str = Field(..., description="The author of the article.")
    keywords: list[str] = Field(..., description="You need to provide me with no
more than 5 keywords.")
    summary: str = Field(..., description="""
    High level summary of the article with relevant facts and details.
    Include all relevant information to provide full picture.
    """)
```

Next, we'll run `mcp-server-fetch` locally, connecting via AutoGen's `StdioServerParams` and `StdioMcpToolAdapter` in main:

```
server_params = StdioServerParams(
    command="python",
    args=["-m", "mcp_server_fetch"],
    read_timeout_seconds=30
)

fetch = await StdioMcpToolAdapter.from_server_params(server_params, "fetch")
```

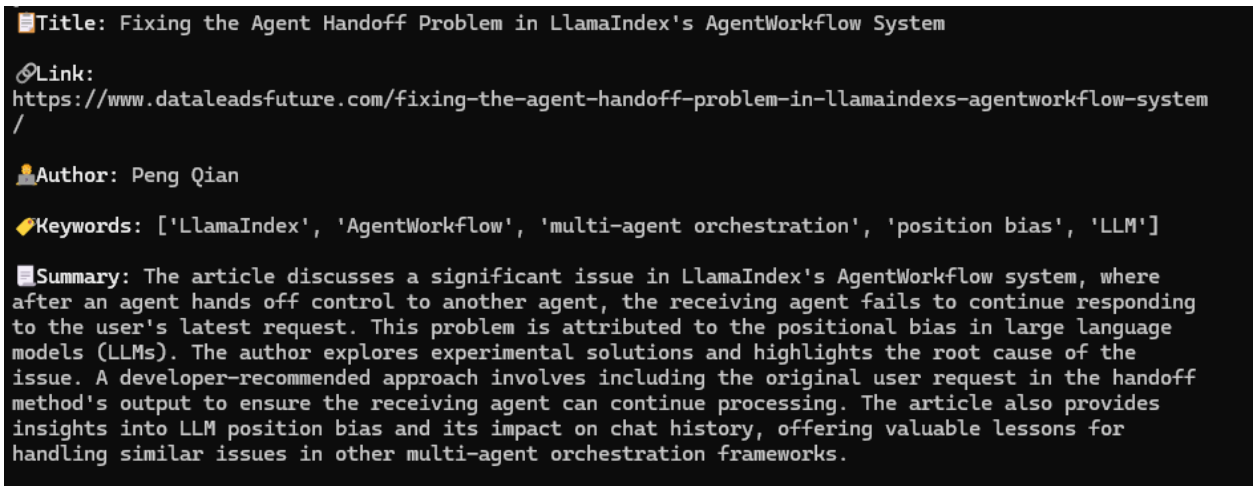
Now define the agent, passing `fetch` mcp tool and our data class. Note: if thinking mode is enabled, `model_client_stream` must be `True` for streaming output. Here we've disabled thinking mode but kept `model_client_stream=True`:

```
agent = AssistantAgent(
    name="web_browser",
    model_client=model_client,
    tools=[fetch],
    system_message="You are a helpful assistant.",
    output_content_type=ArticleDetail,
    model_client_stream=True
)
```

Finally, have the agent read my previous article and produce structured output:

```
result = await Console(
    agent.run_stream(task="""
    Please visit
    https://www.dataleadsfuture.com/fixing-the-agent-handoff-problem-in-llamaindexs-agentworkflow-system/
    and give me a quick summary.
    """)
)

output = cast(ArticleDetail, result.messages[-1].content)

console.print(Markdown(dedent(f"""
\n
\n
**📋Title:** {output.title}

**🔗Link:** {output.url}

**👲Author:** {output.author}

**🏷Keywords:** {output.keywords}

**📃Summary:** {output.summary}
""")))
```

Excellent! The agent successfully fetched the article and generated structured output via `ArticleDetail`:



The agent successfully fetched the article and generated structured output. Image by Author

Our `OpenAILikeChatCompletionClient` implementation hides all structured_output and thinking mode details, making project code beautifully simple.

Conclusion

As Qwen3 deployment grows in enterprises, corresponding agent frameworks must keep pace.
This book explored adapting AutoGen for Qwen3.
Through various approaches, to explained structured_output principles that benefit you even beyond AutoGen.
And also analyzed AutoGen's parameter handling at code level, adding support for Qwen3's `extra_body` parameters.