Last suggestion on backtester\_agent

# In agents/backtester\_agent.py (and similar for other agents needing tools)

from autogen.coding import LocalCommandLineCodeExecutor

from autogen\_core.tools import FunctionTool

import tempfile

# Create a work directory for the executor

work\_dir = os.path.join(project\_root, "agent\_tool\_work\_dir", "backtester") # agent-specific work dir

os.makedirs(work\_dir, exist\_ok=True)

executor = LocalCommandLineCodeExecutor(work\_dir=work\_dir, timeout=120) # 2 min timeout

logging.info(f"BacktesterAgent: Initialized LocalCommandLineCodeExecutor with work\_dir: {work\_dir}")

def execute\_python\_code\_for\_backtest(code\_to\_execute: str) -> str:

"""

Executes the given Python backtesting code, which should include its own data fetching

and print its results (e.g., metrics as a JSON string) to stdout.

Args:

code\_to\_execute (str): The Python code string for the backtest.

Returns:

str: The combined stdout and stderr of the execution.

"""

logger.info(f"Backtester Tool: Attempting to execute backtest code...")

# The executor saves the code to a file in its work\_dir and runs it.

# Ensure the code\_to\_execute is a single block of runnable script.

# The 'python' language identifier tells the executor how to run it.

code\_blocks = [("python", code\_to\_execute)]

try:

execution\_result = executor.execute\_code\_blocks(code\_blocks=code\_blocks)

exit\_code = getattr(execution\_result, 'exit\_code', -1) # Adapt to actual result object

output = getattr(execution\_result, 'output', '')

if exit\_code == 0:

logger.info(f"Backtester Tool: Code executed successfully. Output snippet: {output[:500]}...")

return output # This should be the JSON string of metrics from the script

else:

logger.error(f"Backtester Tool: Code execution failed. Exit code: {exit\_code}. Output: {output}")

return f"Code execution failed with exit code {exit\_code}. Error: {output}"

except Exception as e:

logger.exception("Backtester Tool: Exception during code execution setup or call.")

return f"Exception during code execution tool: {str(e)}"

backtest\_execution\_tool = FunctionTool(

fn=execute\_python\_code\_for\_backtest,

name="run\_python\_backtest\_script", # Clearer name

description="Executes a complete Python script designed for backtesting (e.g., using Backtrader and yfinance). The script must handle its own data fetching and print a JSON string of performance metrics to standard output."

)

TOOL\_AVAILABLE = True # Since we defined it

tools\_for\_backtester = [backtest\_execution\_tool]

**our Proposed Solution - Analysis and Opinion:**

Your idea to have a dedicated "Get Financial Data Agent" (let's call it DataProvisioningAgent for clarity) is a very good one. Here's why and how it aligns with robust system design:

**Advantages:**

1. **Separation of Concerns:**
   * BacktesterAgent focuses solely on running backtests and reporting metrics, assuming valid data is provided.
   * DataProvisioningAgent specializes in fetching, validating, and possibly preprocessing financial data. This is a complex task in itself.
2. **Improved Robustness:**
   * Instead of BacktesterAgent just failing or producing meaningless results with insufficient data, it can identify the data deficiency.
   * The system can then react by invoking the DataProvisioningAgent.
3. **Intelligent Data Fetching:**
   * Your criteria for the DataProvisioningAgent are spot on:
     + **Strategy Type:** Different strategies might benefit from different types or lengths of data (e.g., high-frequency data for scalping DQN vs. daily data for long-term trend-following).
     + **Ticker Specifics:** Volatility and typical cycle lengths vary by asset.
     + **Exhaustive Data:** Covering various market regimes (bullish, bearish, volatile, recent) is crucial for robust strategy evaluation and avoiding overfitting.
4. **Interactive Workflow:**
   * The idea of BacktesterAgent sending a "retry backtest with new data" message after DataProvisioningAgent acts is a good example of a dynamic, collaborative workflow.

**How it could work in the AutoGen Flow:**

1. **Initial Request:** User (via UserProxy) asks to backtest a strategy (e.g., common\_logic/strategies/my\_strategy.py) on a specific ticker. They might or might not provide a data file.
2. **BacktesterAgent's First Attempt:**
   * The Selector chooses BacktesterAgent.
   * BacktesterAgent attempts to use its run\_strategy\_backtest tool.
   * Inside run\_strategy\_backtest (or a pre-check within BacktesterAgent's logic before calling the tool):
     + It checks if data\_file\_path is provided and valid.
     + **Crucially, it would load the data and perform an initial check:**
       - Is the number of data points sufficient for the strategy's parameters (e.g., lookback periods of SMAs, RSI)? Your dummy\_strategy.py has sma\_period and rsi\_period as params. The BacktesterAgent (or its tool) would need access to these to make an informed decision about data length.
       - Are the required columns (Open, High, Low, Close, Volume, Date) present?
3. **Data Deficiency Detected by BacktesterAgent:**
   * If data is insufficient (e.g., "Data length of 25 bars is too short for an SMA period of 50"), the BacktesterAgent does **not** proceed with the full backtest that would error out.
   * Instead, its response message would be something like:  
     "Data in 'data/dummy\_data.csv' is insufficient for strategy 'DummyStrategy' with params {'sma\_period': 50}. Required at least 50 bars, found 25. Requesting more data for ticker AAPL, covering 2022-2023, suitable for a strategy with up to 200-period lookbacks. Next agent should be DataProvisioningAgent."  
     *(It might even suggest a specific date range based on strategy params or default to a longer period like 2-3 years).*
4. **Selector chooses DataProvisioningAgent:**
   * Based on BacktesterAgent's response, the Selector routes the task to DataProvisioningAgent.
5. **DataProvisioningAgent's Action:**
   * It receives the request (ticker, suggested period, maybe strategy type hints).
   * It uses its own tools (e.g., a yfinance\_tool that can download data and save it to a specified path like data/AAPL\_2020-01-01\_2023-12-31.csv).
   * It considers your criteria (volatile periods, etc.) perhaps by:
     + Fetching a longer default period (e.g., 5 years) to ensure various regimes are covered.
     + Having pre-defined "good periods" for certain asset classes or strategy types.
     + Eventually, even using another LLM call to reason about what constitutes "exhaustive data" for the given context.
   * Its response message would be:  
     "Successfully downloaded new data for AAPL (2020-01-01 to 2023-12-31) and saved it to 'data/AAPL\_extended.csv'. BacktesterAgent should retry with this new data file."
6. **Selector chooses BacktesterAgent again:**
   * BacktesterAgent receives the new file path and retries the backtest.

**Implementation Considerations:**

1. **DataProvisioningAgent (agents/data\_provisioning\_agent.py):**
   * Needs a system message defining its role and how to use its tools.
   * Needs a tool, e.g., fetch\_and\_save\_ohlcv\_data(ticker: str, start\_date: str, end\_date: str, output\_file\_path: str) -> str. This tool would use yfinance or another data provider.
2. **Modifications to BacktesterAgent and its Tool (run\_strategy\_backtest):**
   * The run\_strategy\_backtest tool (or logic within BacktesterAgent before calling the tool) needs to perform the data length check against strategy parameters.
     + This implies run\_strategy\_backtest might need to know the strategy's lookback requirements. This could be tricky if parameters are only known inside the strategy file.
     + **Alternative:** load\_strategy\_class\_from\_file could inspect StrategyClass.params to find the largest period if they are defined as class attributes. Or, BacktesterAgent could explicitly ask the user/CodeAgent for maximum lookback period required by the strategy.
   * If data is insufficient, the tool should return a specific structured error or status that BacktesterAgent can interpret to mean "request more data."
   * BacktesterAgent's system message should instruct it on how to formulate a request to DataProvisioningAgent.
3. **Selector Prompt:**
   * Needs to understand the "data insufficient, retry with DataProvisioningAgent" flow.
   * AGENT\_LLM\_MAPPINGS needs an entry for DataProvisioningAgent.

**Regarding your criteria for DataProvisioningAgent:**

* 1. proposed strategy euristic, DQN, ecc.. - Yes, the type of strategy can influence data needs (e.g., tick data for HFT DQN vs. daily for swing). The agent could take a strategy\_type\_hint: str argument.
* 3. ticker ( for different tickers different periods can be significant - Absolutely. The tool needs ticker.
* 2. try to get exhaustive data : volatile periods, bearish periods, bullish periods, recent periods - This is the "intelligent" part.
  + **Simple approach:** Fetch a long default period (e.g., 5-10 years) for daily data.
  + **Advanced:** Could involve analyzing a longer history to identify these regimes and ensure the fetched data covers them. This might even be a sub-task for another specialized analysis agent or an LLM call within DataProvisioningAgent.

This is another fantastic log showing a multi-turn conversation! The system is robustly handling agent selection and LLM calls.

**Analysis of the Log (Focusing on the end and your questions):**

1. **ModelFamily Warning (Q1):**
   * 2025-06-08 12:16:33,200 - WARNING - root - config.<module>:L52 - autogen\_ext.models.ollama.ModelFamily not found. Using dummy.
   * You are absolutely correct that this log line is present in the trace you just provided. It's very near the beginning.
   * **This means that despite previous attempts, the incorrect try-except block for importing ModelFamily from autogen\_ext.models.ollama (which would be around line 52 in your config.py if that line number is accurate from the logger) is *still present and executing* in your config.py.**
   * The successful import INFO - root - config.<module>:L19 - Successfully imported ModelFamily from autogen\_core.models happens, but then later the incorrect one also runs and because it fails, it defines the dummy, potentially overwriting or confusing the ModelFamily namespace.
   * **Action:** You must find and **delete** the try-except block in config.py that attempts to import ModelFamily from autogen\_ext.models.ollama... and logs this warning. The only ModelFamily import should be from autogen\_core.models import ModelFamily.
2. **End of the Log and Termination (Q2):**
   * The log shows a long sequence of successful agent interactions:
     + User -> Selector -> CodeAgent (Ollama)
     + CodeAgent -> Selector -> BacktesterAgent (Gemini)
     + BacktesterAgent -> Selector -> TradingAgent (Ollama)
     + TradingAgent -> Selector -> CodeAgent (Ollama)
     + CodeAgent -> Selector -> BacktesterAgent (Gemini)
     + BacktesterAgent -> Selector -> TradingAgent (Ollama)
     + TradingAgent -> Selector -> **UserProxy** (This is the crucial selection for termination)
       - 2025-06-08 12:17:29,869 - INFO - autogen\_core.events - \_ollama\_client.create:L636 - ... "message": {"role": "assistant", "content": "UserProxy", ...}}
       - 2025-06-08 12:17:29,871 - INFO - autogen\_core - \_single\_threaded\_agent\_runtime.\_process\_publish:L570 - Calling message handler for UserProxy... with message type GroupChatRequestPublish...
   * **Soliciting User Input:**
     + 2025-06-08 12:17:57,811 - INFO - autogen\_core - \_single\_threaded\_agent\_runtime.publish\_message:L404 - Publishing message of type GroupChatMessage to all subscribers: {'message': UserInputRequestedEvent(source='UserProxy', ... content='', ...)}
     + This UserInputRequestedEvent is the mechanism by which the UserProxyAgent (from autogen-agentchat) signals that it needs human input. Your Console UI component should detect this (or a similar event/state) and print "Enter your response:" to the actual console/terminal.
   * **The log snippet you provided ends *while the system is waiting for user input*.**

Enter your response: 2025-06-08 11:08:57,812 - INFO - autogen\_core.events - \_single\_threaded\_agent\_runtime.\_process\_publish:L573 - {"payload": "{\"message\":{\"source\":\"UserProxy\",\"models\_usage\":null,\"metadata\":{},\"created\_at\":\"2025-06-08T08:08:57.811361Z\"}}", "sender": "UserProxy\_a25e36dc-f8e7-42c6-882a-6ee153e2d851/a25e36dc-f8e7-42c6-882a-6ee153e2d851", "receiver": null, "kind": "MessageKind.PUBLISH", "delivery\_stage": "DeliveryStage.DELIVER", "type": "Message"}

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).

The "Enter your response:" part is typically printed to sys.stdout by the UserProxyAgent's input solicitation mechanism (or the Console reacting to UserInputRequestedEvent). The log lines after it are just further event processing within AutoGen.

* + **Why it "stopped":** You mentioned you "stopped it brutally from the ide". The system was correctly paused, waiting for you to type something into the console where the script was running. If you had typed TERMINATE and pressed Enter, your TextMentionTermination("TERMINATE") condition on the SelectorGroupChat would have caught it, and the chat would have ended gracefully.

1. **Agent Responses & Selector Behavior:**
   * The agents are generating very long, detailed responses, often including full code snippets and explanations.
   * The Selector (using gemma3:1b) sometimes fails to output just an agent name and instead outputs code or longer text, requiring the SelectorGroupChat to re-prompt it. This is evident in the logs like:
     + DEBUG - autogen\_agentchat - \_selector\_group\_chat.\_select\_speaker:L277 - Model failed to select a valid name: ```python ... ``` (attempt 1)
     + Followed by a successful selection on attempt 2 after the re-prompt.
   * This indicates the prompt for the Selector LLM needs to be even more strict, or gemma3:1b isn't great at following very constrained output format instructions when the context (history) is very long and full of code.

**No New Exceptions in This Log:**  
Crucially, this log snippet **does not show any new Python exceptions** that would have crashed the program. The "brutal stop" was external.

**Summary & Next Steps:**

1. ✅ **ModelFamily Import (Still the #1 Action):**
   * Open config.py.
   * **Search for and DELETE the try-except block around line 52 that tries to import ModelFamily from autogen\_ext.models.ollama (or autogen\_ext.models.ollama.\_types).** This is causing the "Using dummy" warning and means your Ollama clients are getting model\_info={'family': 'R1\_dummy\_...'}.
   * Ensure your primary import from autogen\_core.models import ModelFamily (around line 19) is active and that AGENT\_LLM\_MAPPINGS uses ModelFamily.R1 (the enum member).
   * **Verify:** After this change, run the script. The "Using dummy" warning for ModelFamily MUST be gone. The model\_info logged for Ollama clients should show something like {'family': <ModelFamily.R1: 'r1'>, ...} (the enum representation) instead of just {'family': 'r1', ...}. This is important because the OllamaChatCompletionClient might behave differently or more correctly with the actual enum.
2. **Logging Visibility:**
   * The detailed logs are appearing, which is good. Your logging.basicConfig in main\_autogen.py is working.
3. **Selector Prompt:**
   * Your refined selector\_prompt\_template (the one that relies more on {history} and less on explicit {last\_speaker\_name} which caused the KeyError) is working for the first turn.
   * However, the Selector LLM still sometimes outputs more than just the agent name. You can try making the instruction "Your response MUST be ONLY the exact name of ONE agent from the list..." even more prominent, perhaps repeating it or using ALL CAPS more.
   * Consider what the list of available agents for the Selector is at each step. The log shows:
     + Initial: ['UserProxy', 'CodeAgent', 'BacktesterAgent', 'TradingAgent'] (BO\_Agent is in your mapping but not in the list passed to Selector initially).
     + After CodeAgent replies: ['UserProxy', 'BacktesterAgent', 'TradingAgent'] (CodeAgent is removed as it just spoke).
     + And so on. This is correct behavior from SelectorGroupChat.
4. **Agent System Messages & Functionality:**
   * This is where most of the work lies now. The agents are "talking" but not yet *doing* much beyond text generation.
   * **CodeAgent:** Its output is very verbose and includes a lot of conceptual explanation instead of just clean, runnable strategy code. Its system message needs to be much stricter: "You are a Python code generation expert. Your *only* output should be a Python code block containing a complete, runnable Backtrader strategy class. Do not include explanations, example usage, or any text outside the code block unless specifically asked. The strategy class should be named MyGeneratedStrategy. Save the strategy to 'common\_logic/strategies/generated\_strategy.py' and parameters to 'common\_logic/strategies/generated\_strategy\_params.json'. State these file paths in your response."
   * **BacktesterAgent:**
     + Currently, it's just receiving text/code and using Gemini to analyze it.
     + **Tool Implementation:** You need to fix the PythonCodeExecutionTool import/definition in agents/backtester\_agent.py.
       - Change from autogen\_ext.tools.\_code\_execution import PythonCodeExecutionTool to something valid. A good starting point for a safe executor:
       - # agents/backtester\_agent.py
       - from autogen.coding import LocalCommandLineCodeExecutor
       - from autogen\_core.tools import FunctionTool
       - import tempfile
       - import shutil
       - # Create an executor instance
       - # You might want to configure a work\_dir that gets cleaned up
       - temp\_dir = tempfile.mkdtemp()
       - executor = LocalCommandLineCodeExecutor(work\_dir=temp\_dir) # timeout=60
       - logging.info(f"BacktesterAgent: Initialized LocalCommandLineCodeExecutor with work\_dir: {temp\_dir}")
       - def execute\_python\_code\_for\_backtest(code\_to\_execute: str, file\_name: str = "strategy\_to\_run.py") -> str:
       - """
       - Saves the given Python code to a file and executes it.
       - Returns the combined stdout and stderr of the execution.
       - The code should print its results (e.g., backtest metrics as JSON) to stdout.
       - Args:
       - code\_to\_execute (str): The Python code string to execute.
       - file\_name (str, optional): The name of the file to save the code to.
       - """
       - logger.info(f"Backtester Tool: Attempting to execute code saved to {file\_name}...")
       - try:
       - # The executor saves the code to a file in its work\_dir
       - execution\_result = executor.execute\_code\_blocks(
       - code\_blocks=[("python", code\_to\_execute)] # Expects list of (language, code)
       - )
       - # execution\_result will be a CodeExecutionResult object
       - # result\_str = f"Exit Code: {execution\_result.exit\_code}\nOutput:\n{execution\_result.output}"
       - if execution\_result.exit\_code == 0:
       - logger.info(f"Backtester Tool: Code executed successfully. Output: {execution\_result.output[:500]}...")
       - return execution\_result.output # This should be the JSON string of metrics
       - else:
       - logger.error(f"Backtester Tool: Code execution failed. Exit code: {execution\_result.exit\_code}. Output: {execution\_result.output}")
       - return f"Code execution failed with exit code {execution\_result.exit\_code}. Error: {execution\_result.output}"
       - except Exception as e:
       - logger.exception("Backtester Tool: Exception during code execution.")
       - return f"Exception during code execution: {str(e)}"
       - # Create the FunctionTool
       - backtest\_execution\_tool = FunctionTool(
       - fn=execute\_python\_code\_for\_backtest,
       - name="execute\_python\_backtest",
       - description="Saves Python backtesting code (e.g., using Backtrader) to a file and runs it. The code MUST print its results as a JSON string to stdout."
       - )
       - tools\_for\_backtester = [backtest\_execution\_tool]

TOOL\_AVAILABLE = True # Assuming executor setup is successful

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

* + - Its system message should then instruct it: "You will receive Python code for a trading strategy. Your task is to use the execute\_python\_backtest tool to run this code. The code should be designed to print a JSON string of backtest metrics to stdout. After execution, present these metrics."
  + **backtrader\_runner.py as the executed code:** The code that CodeAgent generates and BacktesterAgent executes should be a self-contained script that uses your backtrader\_runner.run\_backtest function and then print(json.dumps(results)) at the end.

This is a lot, but the key is to:

1. Fix the ModelFamily import.
2. Focus on getting the *first* tool (execute\_python\_backtest for BacktesterAgent) working.

The system is behaving much more predictably now!

**ddressing Your Questions:**

**Q1: ModelFamily Warning (WARNING:root:config.<module>:L52 - autogen\_ext.models.ollama.ModelFamily not found. Using dummy.)**

* **Yes, this warning is still present in your log:**  
  2025-06-08 12:16:33,200 - WARNING - root - config.<module>:L52 - autogen\_ext.models.ollama.ModelFamily not found. Using dummy.
* **Why it's there:** You have two try-except blocks attempting to import ModelFamily in your config.py.
  1. The first one (around your original line 19) correctly imports it:  
     2025-06-08 12:16:32,696 - INFO - root - config.<module>:L19 - Successfully imported ModelFamily from autogen\_core.models
  2. The second one (around your original line 52) tries to import it from an incorrect path (autogen\_ext.models.ollama), fails, logs the warning, and defines a dummy ModelFamily class. This dummy class's R1 attribute is just a string.
* **Impact:** When your AGENT\_LLM\_MAPPINGS use ModelFamily.R1, they are picking up the *dummy* string value because the second (failing) import block redefines ModelFamily in the config.py's scope.
  1. This is why you see {'family': 'r1', ...} (string 'r1') in the model\_info for Ollama clients in the logs, instead of an enum representation like <ModelFamily.R1: 'r1'>.
  2. The OllamaChatCompletionClient from autogen-ext is tolerant enough to accept the string 'r1' for the family (or it defaults if it doesn't recognize the string), so it *doesn't crash* during initialization anymore. This is good luck, but not ideal.
* **ACTION: Open config.py. Find the try-except block that produces the "autogen\_ext.models.ollama.ModelFamily not found. Using dummy." warning (it's the one trying to import from autogen\_ext.models.ollama... around your original line 52). DELETE this entire try-except block. The import from autogen\_core.models is the only one you need.**

**Q2: The end of the log is not very clear to me. Who terminated and why did everything stop?**

* The log you provided **does not show a crash or an exception at the end.**
* It shows a very long sequence of successful agent turns. Let's trace the last few:
  1. ... Many turns ...
  2. Selector (Ollama) picks CodeAgent. (Log timestamp 2025-06-08 12:17:23,241)
  3. CodeAgent (Ollama) responds with a very long message containing Python code and explanations. (Log timestamp 2025-06-08 12:17:23,276 for the LLM call, response displayed around 2025-06-08 12:17:31,458)
  4. Selector (Ollama) is called again. It needs to decide who speaks after CodeAgent.
     + Its LLM (gemma3:1b) is called.
     + 2025-06-08 12:17:31,462 - DEBUG - httpcore.http11 - \_trace.atrace:L87 - receive\_response\_headers.started request=<Request [b'POST']>  
       This is the last line of your log.
* **Why it "stopped":** The program was in the middle of the Selector agent making an LLM call to Ollama to decide the next speaker. You mentioned, *"i stopped it brutally from the ide - the finishing part should be dirty"*. This is exactly what happened. The script was running; it hadn't terminated on its own or crashed. It was waiting for the HTTP response from the Ollama server for the Selector's LLM call.
* **If you hadn't stopped it:**
  1. Ollama would have returned a response (e.g., "BacktesterAgent").
  2. The Selector would have published this choice.
  3. The BacktesterAgent would have been called.
  4. The conversation would have continued until your MaxMessageTermination(30) limit was hit, or until an agent said "TERMINATE" and your TextMentionTermination condition was met.

**Other Important Observations from the Log:**

* **Logging is working!** Your logging.basicConfig in main\_autogen.py is effective, and you are seeing INFO and DEBUG messages.
* **Successful Client Initializations:** All agents that need LLM clients are getting them.
  + CodeAgent (Ollama) - OK
  + BacktesterAgent (Gemini) - OK (Tools are disabled due to placeholder PythonCodeExecutionTool as expected for now: WARNING - agents.backtester\_agent - backtester\_agent.<module>:L134 - BacktesterAgent (model\_client: UnknownClient): Client reports function\_calling=True, but PythonCodeExecutionTool or executor not available. Tools disabled.)
  + TradingAgent (Ollama) - OK
  + Selector (Ollama) - OK
  + UserProxy (LLM-less) - OK (The ERROR log about it receiving a client object but then initializing without an LLM is your correct handling in user\_proxy\_agent.py).
* **Agent Communication is Happening:** Agents are taking turns, and their messages (though sometimes very verbose and not perfectly on task) are being passed around and displayed.
* **Selector Prompt Needs Work:** The Selector (using gemma3:1b) is still sometimes outputting full code blocks or conversational text instead of *just* the agent name. This forces the SelectorGroupChat to re-prompt it.
  + 2025-06-08 12:17:23,241 - DEBUG - autogen\_agentchat - \_selector\_group\_chat.\_select\_speaker:L277 - Model failed to select a valid name: ```python ... ``` (attempt 1)
  + 2025-06-08 12:17:24,110 - DEBUG - autogen\_agentchat - \_selector\_group\_chat.\_select\_speaker:L300 - Model selected a valid name: TradingAgent (attempt 2)  
    Your refined selector prompt from the previous session helped, but gemma3:1b might still struggle with very strict output formatting when given long conversational histories.

**Immediate Next Steps (After Your Break):**

1. **CRITICAL: Fix ModelFamily Import in config.py:**
   * Open config.py.
   * **DELETE** the entire try-except block that attempts to import ModelFamily from autogen\_ext.models.ollama... (the one causing the "Using dummy" warning). This is the block that was around your original line 52.
   * The import from autogen\_core.models import ModelFamily should be the *only* one defining ModelFamily.
   * Verify that AGENT\_LLM\_MAPPINGS uses ModelFamily.R1 (the actual enum member) for the family key in all ollama\_model\_info dictionaries.
2. **Refine Selector Prompt (in main\_autogen.py):**  
   While your previous refinement helped, try to make it even more explicit if gemma3:1b continues to misbehave.
3. selector\_prompt\_template = """You are an AI orchestrator. Your ONLY task is to select the next agent.
4. Available agents: {participants}
5. Their roles:
6. {roles}
7. Conversation History (newest is last):
8. {history}
9. Based ONLY on the last few messages and the overall goal, output the EXACT NAME of the single most appropriate agent from the list {participants} to speak next.
10. Your entire response MUST be just the agent's name and nothing else. For example:
11. CodeAgent
12. DO NOT add any explanations, punctuation, or other text.
13. Select the next agent to act:

"""

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

This simplifies it and tries to be very direct.

1. **Address Tool Placeholders (e.g., in backtester\_agent.py):**
   * ERROR - root - backtester\_agent.<module>:L58 - Could not import actual tools/memory for BacktesterAgent: No module named 'autogen\_ext.tools.\_code\_execution'. Using placeholders.
   * You need to provide a real PythonCodeExecutionTool (or similar) and executor. As discussed, a good starting point for a safer executor is autogen.coding.LocalCommandLineCodeExecutor.
   * # In agents/backtester\_agent.py (and similar for other agents needing tools)
   * from autogen.coding import LocalCommandLineCodeExecutor
   * from autogen\_core.tools import FunctionTool
   * import tempfile
   * # Create a work directory for the executor
   * work\_dir = os.path.join(project\_root, "agent\_tool\_work\_dir", "backtester") # agent-specific work dir
   * os.makedirs(work\_dir, exist\_ok=True)
   * executor = LocalCommandLineCodeExecutor(work\_dir=work\_dir, timeout=120) # 2 min timeout
   * logging.info(f"BacktesterAgent: Initialized LocalCommandLineCodeExecutor with work\_dir: {work\_dir}")
   * def execute\_python\_code\_for\_backtest(code\_to\_execute: str) -> str:
   * """
   * Executes the given Python backtesting code, which should include its own data fetching
   * and print its results (e.g., metrics as a JSON string) to stdout.
   * Args:
   * code\_to\_execute (str): The Python code string for the backtest.
   * Returns:
   * str: The combined stdout and stderr of the execution.
   * """
   * logger.info(f"Backtester Tool: Attempting to execute backtest code...")
   * # The executor saves the code to a file in its work\_dir and runs it.
   * # Ensure the code\_to\_execute is a single block of runnable script.
   * # The 'python' language identifier tells the executor how to run it.
   * code\_blocks = [("python", code\_to\_execute)]
   * try:
   * execution\_result = executor.execute\_code\_blocks(code\_blocks=code\_blocks)
   * exit\_code = getattr(execution\_result, 'exit\_code', -1) # Adapt to actual result object
   * output = getattr(execution\_result, 'output', '')
   * if exit\_code == 0:
   * logger.info(f"Backtester Tool: Code executed successfully. Output snippet: {output[:500]}...")
   * return output # This should be the JSON string of metrics from the script
   * else:
   * logger.error(f"Backtester Tool: Code execution failed. Exit code: {exit\_code}. Output: {output}")
   * return f"Code execution failed with exit code {exit\_code}. Error: {output}"
   * except Exception as e:
   * logger.exception("Backtester Tool: Exception during code execution setup or call.")
   * return f"Exception during code execution tool: {str(e)}"
   * backtest\_execution\_tool = FunctionTool(
   * fn=execute\_python\_code\_for\_backtest,
   * name="run\_python\_backtest\_script", # Clearer name
   * description="Executes a complete Python script designed for backtesting (e.g., using Backtrader and yfinance). The script must handle its own data fetching and print a JSON string of performance metrics to standard output."
   * )
   * TOOL\_AVAILABLE = True # Since we defined it

tools\_for\_backtester = [backtest\_execution\_tool]

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

Then, in BacktesterAgent's system message, instruct it to use run\_python\_backtest\_script and that the code provided to this tool should be a complete script that prints JSON results.

You've successfully got the agents talking! Now it's about making those conversations more productive and enabling them to *do* things with tools. The ModelFamily fix is the immediate next step to ensure client stability.

So far we have achieved :

* **Logging is working!** Your logging.basicConfig in main\_autogen.py is effective, and you are seeing INFO and DEBUG messages.
* **Successful Client Initializations:** All agents that need LLM clients are getting them.
  + CodeAgent (Ollama) - OK
  + BacktesterAgent (Gemini) - OK (Tools are disabled due to placeholder PythonCodeExecutionTool as expected for now: WARNING - agents.backtester\_agent - backtester\_agent.<module>:L134 - BacktesterAgent (model\_client: UnknownClient): Client reports function\_calling=True, but PythonCodeExecutionTool or executor not available. Tools disabled.)
  + TradingAgent (Ollama) - OK
  + Selector (Ollama) - OK
  + UserProxy (LLM-less) - OK (The ERROR log about it receiving a client object but then initializing without an LLM is your correct handling in user\_proxy\_agent.py).
* **Agent Communication is Happening:** Agents are taking turns, and their messages (though sometimes very verbose and not perfectly on task) are being passed around and displayed.
* **Selector Prompt Needs Work:** The Selector (using gemma3:1b) is still sometimes outputting full code blocks or conversational text instead of *just* the agent name. This forces the SelectorGroupChat to re-prompt it.
  + 2025-06-08 12:17:23,241 - DEBUG - autogen\_agentchat - \_selector\_group\_chat.\_select\_speaker:L277 - Model failed to select a valid name: ```python ... ``` (attempt 1)
  + 2025-06-08 12:17:24,110 - DEBUG - autogen\_agentchat - \_selector\_group\_chat.\_select\_speaker:L300 - Model selected a valid name: TradingAgent (attempt 2)  
    Your refined selector prompt from the previous session helped, but gemma3:1b might still struggle with very strict output formatting when given long conversational histories.

**Refine Selector Prompt (in main\_autogen.py):**  
While your previous refinement helped, try to make it even more explicit if gemma3:1b continues to misbehave.

selector\_prompt\_template = """You are an AI orchestrator. Your ONLY task is to select the next agent.

1. Available agents: {participants}
2. Their roles:
3. {roles}
4. Conversation History (newest is last):
5. {history}
6. Based ONLY on the last few messages and the overall goal, output the EXACT NAME of the single most appropriate agent from the list {participants} to speak next.
7. Your entire response MUST be just the agent's name and nothing else. For example:
8. CodeAgent
9. DO NOT add any explanations, punctuation, or other text.
10. Select the next agent to act:
11. """

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

This simplifies it and tries to be very direct.

1. **Address Tool Placeholders (e.g., in backtester\_agent.py):**
   * ERROR - root - backtester\_agent.<module>:L58 - Could not import actual tools/memory for BacktesterAgent: No module named 'autogen\_ext.tools.\_code\_execution'. Using placeholders.
   * You need to provide a real PythonCodeExecutionTool (or similar) and executor. As discussed, a good starting point for a safer executor is autogen.coding.LocalCommandLineCodeExecutor.
   * # In agents/backtester\_agent.py (and similar for other agents needing tools)
   * from autogen.coding import LocalCommandLineCodeExecutor
   * from autogen\_core.tools import FunctionTool
   * import tempfile
   * # Create a work directory for the executor
   * work\_dir = os.path.join(project\_root, "agent\_tool\_work\_dir", "backtester") # agent-specific work dir
   * os.makedirs(work\_dir, exist\_ok=True)
   * executor = LocalCommandLineCodeExecutor(work\_dir=work\_dir, timeout=120) # 2 min timeout
   * logging.info(f"BacktesterAgent: Initialized LocalCommandLineCodeExecutor with work\_dir: {work\_dir}")
   * def execute\_python\_code\_for\_backtest(code\_to\_execute: str) -> str:
   * """
   * Executes the given Python backtesting code, which should include its own data fetching
   * and print its results (e.g., metrics as a JSON string) to stdout.
   * Args:
   * code\_to\_execute (str): The Python code string for the backtest.
   * Returns:
   * str: The combined stdout and stderr of the execution.
   * """
   * logger.info(f"Backtester Tool: Attempting to execute backtest code...")
   * # The executor saves the code to a file in its work\_dir and runs it.
   * # Ensure the code\_to\_execute is a single block of runnable script.
   * # The 'python' language identifier tells the executor how to run it.
   * code\_blocks = [("python", code\_to\_execute)]
   * try:
   * execution\_result = executor.execute\_code\_blocks(code\_blocks=code\_blocks)
   * exit\_code = getattr(execution\_result, 'exit\_code', -1) # Adapt to actual result object
   * output = getattr(execution\_result, 'output', '')
   * if exit\_code == 0:
   * logger.info(f"Backtester Tool: Code executed successfully. Output snippet: {output[:500]}...")
   * return output # This should be the JSON string of metrics from the script
   * else:
   * logger.error(f"Backtester Tool: Code execution failed. Exit code: {exit\_code}. Output: {output}")
   * return f"Code execution failed with exit code {exit\_code}. Error: {output}"
   * except Exception as e:
   * logger.exception("Backtester Tool: Exception during code execution setup or call.")
   * return f"Exception during code execution tool: {str(e)}"
   * backtest\_execution\_tool = FunctionTool(
   * fn=execute\_python\_code\_for\_backtest,
   * name="run\_python\_backtest\_script", # Clearer name
   * description="Executes a complete Python script designed for backtesting (e.g., using Backtrader and yfinance). The script must handle its own data fetching and print a JSON string of performance metrics to standard output."
   * )
   * TOOL\_AVAILABLE = True # Since we defined it

tools\_for\_backtester = [backtest\_execution\_tool]

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

Then, in BacktesterAgent's system message, instruct it to use run\_python\_backtest\_script and that the code provided to this tool should be a complete script that prints JSON results.

You've successfully got the agents talking! Now it's about making those conversations more productive and enabling them to *do* things with tools.