GroupChat Swarm Documentation

A production-grade multi-agent system enabling sophisticated group conversations between AI agents with customizable speaking patterns, parallel processing capabilities, and comprehensive conversation tracking.

Advanced Configuration

Agent Parameters

Parameter	Type	Default	Description
agent_name	str	Required	Unique identifier for the agent
system_prompt	str	Required	Role and behavior instructions
llm	Any	Required	Language model instance
max_loops	int	1	Maximum conversation turns
autosave	bool	False	Enable conversation saving
dashboard	bool	False	Enable monitoring dashboard
verbose	bool	True	Enable detailed logging
dynamic_temperature	bool	True	Enable dynamic temperature
retry_attempts	int	1	Failed request retry count
context_length	int	200000	Maximum context window
output_type	str	"string"	Response format type
streaming_on	bool	False	Enable streaming responses

GroupChat Parameters

| Parameter | Type | Default | Description |

|-----------|------|---------|-------------|

| name | str | "GroupChat" | Chat group identifier |

| description | str | "" | Purpose description |

| agents | List[Agent] | [] | Participating agents |

| speaker_fn | Callable | round_robin | Speaker selection function |

| max_loops | int | 10 | Maximum conversation turns |

Table of Contents

Installation

```bash
pip3 install swarms swarm-models loguru
```

```

## Core Concepts

The GroupChat system consists of several key components:

1. **Agents**: Individual AI agents with specialized knowledge and roles
2. **Speaker Functions**: Control mechanisms for conversation flow
3. **Chat History**: Structured conversation tracking
4. **Response Models**: Pydantic models for data validation

## Basic Usage

```python

import os
from dotenv import load_dotenv
from swarm_models import OpenAIChat
from swarms import Agent, GroupChat, expertise_based


if __name__ == "__main__":

    load_dotenv()

    # Get the OpenAI API key from the environment variable

```
api_key = os.getenv("OPENAI_API_KEY")

# Create an instance of the OpenAIChat class
model = OpenAIChat(
    openai_api_key=api_key,
    model_name="gpt-4o-mini",
    temperature=0.1,
)

# Example agents
agent1 = Agent(
    agent_name="Financial-Analysis-Agent",
    system_prompt="You are a financial analyst specializing in investment strategies.",
    llm=model,
    max_loops=1,
    autosave=False,
    dashboard=False,
    verbose=True,
    dynamic_temperature_enabled=True,
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    output_type="string",
    streaming_on=False,
)
```

```
    agent2 = Agent(
        agent_name="Tax-Adviser-Agent",
        system_prompt="You are a tax adviser who provides clear and concise guidance on tax-related
queries.",
        llm=model,
        max_loops=1,
        autosave=False,
        dashboard=False,
        verbose=True,
        dynamic_temperature_enabled=True,
        user_name="swarms_corp",
        retry_attempts=1,
        context_length=200000,
        output_type="string",
        streaming_on=False,
    )

    agents = [agent1, agent2]

    chat = GroupChat(
        name="Investment Advisory",
        description="Financial and tax analysis group",
        agents=agents,
        speaker_fn=expertise_based,
    )
```

```
    history = chat.run(
        "How to optimize tax strategy for investments?"
    )
    print(history.model_dump_json(indent=2))
```

Speaker Functions

Built-in Functions

```python
def round_robin(history: List[str], agent: Agent) -> bool:
    """
    Enables agents to speak in turns.
    Returns True for each agent in sequence.
    """
    return True

def expertise_based(history: List[str], agent: Agent) -> bool:
    """
    Enables agents to speak based on their expertise.
    Returns True if agent's role matches conversation context.
    """
    return agent.system_prompt.lower() in history[-1].lower() if history else True
```

```
def random_selection(history: List[str], agent: Agent) -> bool:
    """
    Randomly selects speaking agents.
    Returns True/False with 50% probability.
    """
    import random
    return random.choice([True, False])

def most_recent(history: List[str], agent: Agent) -> bool:
    """
    Enables agents to respond to their mentions.
    Returns True if agent was last speaker.
    """
    return agent.agent_name == history[-1].split(":")[0].strip() if history else True
```

Custom Speaker Function Example

```python
def custom_speaker(history: List[str], agent: Agent) -> bool:
    """
    Custom speaker function with complex logic.

    Args:
        history: Previous conversation messages
```

```
    agent: Current agent being evaluated

Returns:
    bool: Whether agent should speak
"""
# No history - let everyone speak
if not history:
    return True

last_message = history[-1].lower()

# Check for agent expertise keywords
expertise_relevant = any(
    keyword in last_message
    for keyword in agent.expertise_keywords
)

# Check for direct mentions
mentioned = agent.agent_name.lower() in last_message

# Check if agent hasn't spoken recently
not_recent_speaker = not any(
    agent.agent_name in msg
    for msg in history[-3:]
)
```

```
    return expertise_relevant or mentioned or not_recent_speaker

# Usage
chat = GroupChat(
    agents=[agent1, agent2],
    speaker_fn=custom_speaker
)
```

Response Models

Complete Schema

```python
class AgentResponse(BaseModel):
    """Individual agent response in a conversation turn"""
    agent_name: str
    role: str
    message: str
    timestamp: datetime = Field(default_factory=datetime.now)
    turn_number: int
    preceding_context: List[str] = Field(default_factory=list)

class ChatTurn(BaseModel):
    """Single turn in the conversation"""
    turn_number: int
```

```
    responses: List[AgentResponse]
    task: str
    timestamp: datetime = Field(default_factory=datetime.now)

class ChatHistory(BaseModel):
    """Complete conversation history"""
    turns: List[ChatTurn]
    total_messages: int
    name: str
    description: str
    start_time: datetime = Field(default_factory=datetime.now)
```

Advanced Examples

Multi-Agent Analysis Team

```python
# Create specialized agents
data_analyst = Agent(
    agent_name="Data-Analyst",
    system_prompt="You analyze numerical data and patterns",
    llm=model
)

market_expert = Agent(
```

```
    agent_name="Market-Expert",
    system_prompt="You provide market insights and trends",
    llm=model
)

strategy_advisor = Agent(
    agent_name="Strategy-Advisor",
    system_prompt="You formulate strategic recommendations",
    llm=model
)

# Create analysis team
analysis_team = GroupChat(
    name="Market Analysis Team",
    description="Comprehensive market analysis group",
    agents=[data_analyst, market_expert, strategy_advisor],
    speaker_fn=expertise_based,
    max_loops=15
)

# Run complex analysis
history = analysis_team.run("""
    Analyze the current market conditions:
    1. Identify key trends
    2. Evaluate risks
    3. Recommend investment strategy
```

""")
```

### Parallel Processing

```python
# Define multiple analysis tasks
tasks = [
    "Analyze tech sector trends",
    "Evaluate real estate market",
    "Review commodity prices",
    "Assess global economic indicators"
]

# Run tasks concurrently
histories = chat.concurrent_run(tasks)

# Process results
for task, history in zip(tasks, histories):
    print(f"\nAnalysis for: {task}")
    for turn in history.turns:
        for response in turn.responses:
            print(f"{response.agent_name}: {response.message}")
```

Best Practices
```

1. **Agent Design**
   - Give agents clear, specific roles
   - Use detailed system prompts
   - Set appropriate context lengths
   - Enable retries for reliability

2. **Speaker Functions**
   - Match function to use case
   - Consider conversation flow
   - Handle edge cases
   - Add appropriate logging

3. **Error Handling**
   - Use try-except blocks
   - Log errors appropriately
   - Implement retry logic
   - Provide fallback responses

4. **Performance**
   - Use concurrent processing for multiple tasks
   - Monitor context lengths
   - Implement proper cleanup
   - Cache responses when appropriate

API Reference

GroupChat Methods

Method	Description	Arguments	Returns
run	Run single conversation	task: str	ChatHistory
batched_run	Run multiple sequential tasks	tasks: List[str]	List[ChatHistory]
concurrent_run	Run multiple parallel tasks	tasks: List[str]	List[ChatHistory]
get_recent_messages	Get recent messages	n: int = 3	List[str]

Agent Methods

Method	Description	Returns
run	Process single task	str
generate_response	Generate LLM response	str
save_context	Save conversation context	None