# `Agent` Documentation

Swarm Agent is a powerful autonomous agent framework designed to connect Language Models (LLMs) with various tools and long-term memory. This class provides the ability to ingest and process various types of documents such as PDFs, text files, Markdown files, JSON files, and more. The Agent structure offers a wide range of features to enhance the capabilities of LLMs and facilitate efficient task execution.

1. **Conversational Loop**: It establishes a conversational loop with a language model. This means it allows you to interact with the model in a back-and-forth manner, taking turns in the conversation.

2. **Feedback Collection**: The class allows users to provide feedback on the responses generated by the model. This feedback can be valuable for training and improving the model's responses over time.

3. **Stoppable Conversation**: You can define custom stopping conditions for the conversation, allowing you to stop the interaction based on specific criteria. For example, you can stop the conversation if a certain keyword is detected in the responses.

4. **Retry Mechanism**: The class includes a retry mechanism that can be helpful if there are issues generating responses from the model. It attempts to generate a response multiple times before raising an error.

## Architecture

```mermaid
```

```
graph TD
    A[Task Initiation] -->|Receives Task| B[Initial LLM Processing]
    B -->|Interprets Task| C[Tool Usage]
    C -->|Calls Tools| D[Function 1]
    C -->|Calls Tools| E[Function 2]
    D -->|Returns Data| C
    E -->|Returns Data| C
    C -->|Provides Data| F[Memory Interaction]
    F -->|Stores and Retrieves Data| G[RAG System]
    G -->|ChromaDB/Pinecone| H[Enhanced Data]
    F -->|Provides Enhanced Data| I[Final LLM Processing]
    I -->|Generates Final Response| J[Output]
    C -->|No Tools Available| K[Skip Tool Usage]
    K -->|Proceeds to Memory Interaction| F
    F -->|No Memory Available| L[Skip Memory Interaction]
    L -->|Proceeds to Final LLM Processing| I
```

### `Agent` Attributes

Attribute	Description
`id`	A unique identifier for the agent instance.
`llm`	The language model instance used by the agent.
`template`	The template used for formatting responses.
`max_loops`	The maximum number of loops the agent can run.

`stopping_condition`	A callable function that determines when the agent should stop looping.
`loop_interval`	The interval (in seconds) between loops.
`retry_attempts`	The number of retry attempts for failed LLM calls.
`retry_interval`	The interval (in seconds) between retry attempts.
`return_history`	A boolean indicating whether the agent should return the conversation history.
`stopping_token`	A token that, when present in the response, stops the agent from looping.
`dynamic_loops`	A boolean indicating whether the agent should dynamically determine the number of loops.
`interactive`	A boolean indicating whether the agent should run in interactive mode.
`dashboard`	A boolean indicating whether the agent should display a dashboard.
`agent_name`	The name of the agent instance.
`agent_description`	A description of the agent instance.
`system_prompt`	The system prompt used to initialize the conversation.
`tools`	A list of callable functions representing tools the agent can use.
`dynamic_temperature_enabled`	A boolean indicating whether the agent should dynamically adjust the temperature of the LLM.
`sop`	The standard operating procedure for the agent.
`sop_list`	A list of strings representing the standard operating procedure.
`saved_state_path`	The file path for saving and loading the agent's state.
`autosave`	A boolean indicating whether the agent should automatically save its state.
`context_length`	The maximum length of the context window (in tokens) for the LLM.
`user_name`	The name used to represent the user in the conversation.
`self_healing_enabled`	A boolean indicating whether the agent should attempt to self-heal in case of errors.
`code_interpreter`	A boolean indicating whether the agent should interpret and execute code snippets.

`multi_modal`	A boolean indicating whether the agent should support multimodal inputs (e.g., text and images).
`pdf_path`	The file path of a PDF document to be ingested.
`list_of_pdf`	A list of file paths for PDF documents to be ingested.
`tokenizer`	An instance of a tokenizer used for token counting and management.
`long_term_memory`	An instance of a `BaseVectorDatabase` implementation for long-term memory management.
`preset_stopping_token`	A boolean indicating whether the agent should use a preset stopping token.
`traceback`	An object used for traceback handling.
`traceback_handlers`	A list of traceback handlers.
`streaming_on`	A boolean indicating whether the agent should stream its responses.
`docs`	A list of document paths or contents to be ingested.
`docs_folder`	The path to a folder containing documents to be ingested.
`verbose`	A boolean indicating whether the agent should print verbose output.
`parser`	A callable function used for parsing input data.
`best_of_n`	An integer indicating the number of best responses to generate (for sampling).
`callback`	A callable function to be called after each agent loop.
`metadata`	A dictionary containing metadata for the agent.
`callbacks`	A list of callable functions to be called during the agent's execution.
`logger_handler`	A handler for logging messages.
`search_algorithm`	A callable function representing the search algorithm for long-term memory retrieval.
`logs_to_filename`	The file path for logging agent activities.
`evaluator`	A callable function used for evaluating the agent's responses.
`output_json`	A boolean indicating whether the agent's output should be in JSON format.

`stopping_func`	A callable function used as a stopping condition for the agent.
`custom_loop_condition`	A callable function used as a custom loop condition for the agent.
`sentiment_threshold`	A float value representing the sentiment threshold for evaluating responses.
`custom_exit_command`	A string representing a custom command for exiting the agent's loop.
`sentiment_analyzer`	A callable function used for sentiment analysis on the agent's outputs.
`limit_tokens_from_string`	A callable function used for limiting the number of tokens in a string.
`custom_tools_prompt`	A callable function used for generating a custom prompt for tool usage.
`tool_schema`	A data structure representing the schema for the agent's tools.
`output_type`	A type representing the expected output type of the agent's responses.
`function_calling_type`	A string representing the type of function calling (e.g., "json").
`output_cleaner`	A callable function used for cleaning the agent's output.
`function_calling_format_type`	A string representing the format type for function calling (e.g., "OpenAI").
`list_base_models`	A list of base models used for generating tool schemas.
`metadata_output_type`	A string representing the output type for metadata.
`state_save_file_type`	A string representing the file type for saving the agent's state (e.g., "json", "yaml").
`chain_of_thoughts`	A boolean indicating whether the agent should use the chain of thoughts technique.
`algorithm_of_thoughts`	A boolean indicating whether the agent should use the algorithm of thoughts technique.
`tree_of_thoughts`	A boolean indicating whether the agent should use the tree of thoughts technique.
`tool_choice`	A string representing the method for tool selection (e.g., "auto").
`execute_tool`	A boolean indicating whether the agent should execute tools.

| `rules` | A string representing the rules for the agent's behavior. |

| `planning` | A boolean indicating whether the agent should perform planning. |

| `planning_prompt` | A string representing the prompt for planning. |

| `device` | A string representing the device on which the agent should run. |

| `custom_planning_prompt` | A string representing a custom prompt for planning. |

| `memory_chunk_size` | An integer representing the maximum size of memory chunks for long-term memory retrieval. |

| `agent_ops_on` | A boolean indicating whether agent operations should be enabled. |

| `return_step_meta` | A boolean indicating whether or not to return JSON of all the steps and additional metadata |

| `output_type` | A Literal type indicating whether to output "string", "str", "list", "json", "dict", "yaml" |

### `Agent` Methods

| Method | Description | Inputs | Usage Example |

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

| `run(task, img=None, *args, **kwargs)` | Runs the autonomous agent loop to complete the given task. | `task` (str): The task to be performed.<br>`img` (str, optional): Path to an image file, if the task involves image processing.<br>`*args`, `**kwargs`: Additional arguments to pass to the language model. | `response = agent.run("Generate a report on financial performance.")` |

| `__call__(task, img=None, *args, **kwargs)` | An alternative way to call the `run` method. | Same as `run`. | `response = agent("Generate a report on financial performance.")` |

| `parse_and_execute_tools(response, *args, **kwargs)` | Parses the agent's response and executes any tools mentioned in it. | `response` (str): The agent's response to be

parsed.<br>`*args`, `**kwargs`: Additional arguments to pass to the tool execution. | `agent.parse_and_execute_tools(response)` |

| `long_term_memory_prompt(query, *args, **kwargs)` | Generates a prompt for querying the agent's long-term memory. | `query` (str): The query to search for in long-term memory.<br>`*args`, `**kwargs`: Additional arguments to pass to the long-term memory retrieval. | `memory_retrieval = agent.long_term_memory_prompt("financial performance")` |

| `add_memory(message)` | Adds a message to the agent's memory. | `message` (str): The message

## Features

- **Language Model Integration**: The Swarm Agent allows seamless integration with different language models, enabling users to leverage the power of state-of-the-art models.
- **Tool Integration**: The framework supports the integration of various tools, enabling the agent to perform a wide range of tasks, from code execution to data analysis and beyond.
- **Long-term Memory Management**: The Swarm Agent incorporates long-term memory management capabilities, allowing it to store and retrieve relevant information for effective decision-making and task execution.
- **Document Ingestion**: The agent can ingest and process various types of documents, including PDFs, text files, Markdown files, JSON files, and more, enabling it to extract relevant information for task completion.
- **Interactive Mode**: Users can interact with the agent in an interactive mode, enabling real-time communication and task execution.

- **Dashboard**: The framework provides a visual dashboard for monitoring the agent's performance and activities.
- **Dynamic Temperature Control**: The Swarm Agent supports dynamic temperature control, allowing for adjustments to the model's output diversity during task execution.
- **Autosave and State Management**: The agent can save its state automatically, enabling seamless resumption of tasks after interruptions or system restarts.
- **Self-Healing and Error Handling**: The framework incorporates self-healing and error-handling mechanisms to ensure robust and reliable operation.
- **Code Interpretation**: The agent can interpret and execute code snippets, expanding its capabilities for tasks involving programming or scripting.
- **Multimodal Support**: The framework supports multimodal inputs, enabling the agent to process and reason about various data types, such as text, images, and audio.
- **Tokenization and Token Management**: The Swarm Agent provides tokenization capabilities, enabling efficient management of token usage and context window truncation.
- **Sentiment Analysis**: The agent can perform sentiment analysis on its generated outputs, allowing for evaluation and adjustment of responses based on sentiment thresholds.
- **Output Filtering and Cleaning**: The framework supports output filtering and cleaning, ensuring that generated responses adhere to specific criteria or guidelines.
- **Asynchronous and Concurrent Execution**: The Swarm Agent supports asynchronous and concurrent task execution, enabling efficient parallelization and scaling of operations.
- **Planning and Reasoning**: The agent can engage in planning and reasoning processes, leveraging techniques such as algorithm of thoughts and chain of thoughts to enhance decision-making and task execution.
- **Agent Operations and Monitoring**: The framework provides integration with agent operations and monitoring tools, enabling real-time monitoring and management of the agent's activities.

## Getting Started

First run the following:

```bash
pip3 install -U swarms
```

And, then now you can get started with the following:

```python
import os
from swarms import Agent
from swarm_models import OpenAIChat
from swarms.prompts.finance_agent_sys_prompt import (
    FINANCIAL_AGENT_SYS_PROMPT,
)

# 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(
    api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
)
```

```
# Initialize the agent
agent = Agent(
    agent_name="Financial-Analysis-Agent_sas_chicken_eej",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    autosave=True,
    dashboard=False,
    verbose=True,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent.json",
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    return_step_meta=False,
    output_type="str",
)


agent.run(
    "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria"
)
print(out)

```

This example initializes an instance of the `Agent` class with an OpenAI language model and a maximum of 3 loops. The `run()` method is then called with a task to generate a report on financial performance, and the agent's response is printed.

## Advanced Usage

The Swarm Agent provides numerous advanced features and customization options. Here are a few examples of how to leverage these features:

### Tool Integration

To integrate tools with the Swarm Agent, you can pass a list of callable functions with types and doc strings to the `tools` parameter when initializing the `Agent` instance. The agent will automatically convert these functions into an OpenAI function calling schema and make them available for use during task execution.

## Requirements for a tool
- Function
  - With types
  - with doc strings

```python
from swarms import Agent
from swarm_models import OpenAIChat
from swarms_memory import ChromaDB
import subprocess
import os

# Making an instance of the ChromaDB class
memory = ChromaDB(
    metric="cosine",
    n_results=3,
    output_dir="results",
    docs_folder="docs",
)

# Model
model = OpenAIChat(
    api_key=os.getenv("OPENAI_API_KEY"),
    model_name="gpt-4o-mini",
    temperature=0.1,
)


# Tools in swarms are simple python functions and docstrings
def terminal(
    code: str,
):
    """
    Run code in the terminal.

    Args:
        code (str): The code to run in the terminal.

    Returns:
        str: The output of the code.
    """
    out = subprocess.run(
        code, shell=True, capture_output=True, text=True
    ).stdout
    return str(out)


def browser(query: str):
    """
    Search the query in the browser with the `browser` tool.

    Args:
        query (str): The query to search in the browser.

    Returns:
        str: The search results.
    """
    import webbrowser

    url = f"https://www.google.com/search?q={query}"
    webbrowser.open(url)
    return f"Searching for {query} in the browser."
```

```
def create_file(file_path: str, content: str):
    """
    Create a file using the file editor tool.

    Args:
        file_path (str): The path to the file.
        content (str): The content to write to the file.

    Returns:
        str: The result of the file creation operation.
    """
    with open(file_path, "w") as file:
        file.write(content)
    return f"File {file_path} created successfully."


def file_editor(file_path: str, mode: str, content: str):
    """
    Edit a file using the file editor tool.

    Args:
        file_path (str): The path to the file.
        mode (str): The mode to open the file in.
        content (str): The content to write to the file.
    Returns:
        str: The result of the file editing operation.
    """
    with open(file_path, mode) as file:
        file.write(content)
    return f"File {file_path} edited successfully."


# Agent
agent = Agent(
    agent_name="Devin",
    system_prompt=(
        "Autonomous agent that can interact with humans and other"
        " agents. Be Helpful and Kind. Use the tools provided to"
        " assist the user. Return all code in markdown format."
    ),
    llm=model,
    max_loops="auto",
    autosave=True,
    dashboard=False,
    streaming_on=True,
    verbose=True,
    stopping_token="<DONE>",
    interactive=True,
    tools=[terminal, browser, file_editor, create_file],
    streaming=True,
    long_term_memory=memory,
)

# Run the agent
out = agent(
    "Create a CSV file with the latest tax rates for C corporations in the following ten states and the District of Columbia: Alabama, California, Florida, Georgia, Illinois, New York, North Carolina, Ohio, Texas, and Washington."
)
print(out)

```

### Long-term Memory Management

The Swarm Agent supports integration with various vector databases for long-term memory management. You can pass an instance of a `BaseVectorDatabase` implementation to the `long_term_memory` parameter when initializing the `Agent`.

```python
import os

from swarms_memory import ChromaDB

from swarms import Agent
from swarm_models import Anthropic
from swarms.prompts.finance_agent_sys_prompt import (
    FINANCIAL_AGENT_SYS_PROMPT,
)

# Initilaize the chromadb client
chromadb = ChromaDB(
    metric="cosine",
    output_dir="fiance_agent_rag",
    # docs_folder="artifacts", # Folder of your documents
)

# Model
model = Anthropic(anthropic_api_key=os.getenv("ANTHROPIC_API_KEY"))


# Initialize the agent
agent = Agent(
    agent_name="Financial-Analysis-Agent",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    agent_description="Agent creates ",
    llm=model,
    max_loops="auto",
    autosave=True,
    dashboard=False,
    verbose=True,
    streaming_on=True,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent.json",
    user_name="swarms_corp",
    retry_attempts=3,
    context_length=200000,
    long_term_memory=chromadb,
)



agent.run(
    "What are the components of a startups stock incentive equity plan"
)

```

### Document Ingestion

The Swarm Agent can ingest various types of documents, such as PDFs, text files, Markdown files, and JSON files. You can pass a list of document paths or contents to the `docs` parameter when initializing the `Agent`.

```python
from swarms.structs import Agent


# Initialize the agent with documents
```

agent = Agent(llm=llm, max_loops=3, docs=["path/to/doc1.pdf", "path/to/doc2.txt"])

### Interactive Mode

The Swarm Agent supports an interactive mode, where users can engage in real-time communication with the agent. To enable interactive mode, set the `interactive` parameter to `True` when initializing the `Agent`.

```python
from swarms.structs import Agent

# Initialize the agent in interactive mode
agent = Agent(llm=llm, max_loops=3, interactive=True)

# Run the agent in interactive mode
agent.interactive_run()
```

### Sentiment Analysis

The Swarm Agent can perform sentiment analysis on its generated outputs using a sentiment analyzer function. You can pass a callable function to the `sentiment_analyzer` parameter when initializing the `Agent`.

```python
```

```
from swarms.structs import Agent
from my_sentiment_analyzer import sentiment_analyzer_function


# Initialize the agent with a sentiment analyzer
agent = Agent(
    agent_name = "sentiment-analyzer-agent-01", system_prompt="..."
    llm=llm, max_loops=3, sentiment_analyzer=sentiment_analyzer_function)
```

### Undo Functionality

```python
# Feature 2: Undo functionality
response = agent.run("Another task")
print(f"Response: {response}")
previous_state, message = agent.undo_last()
print(message)
```

### Response Filtering

```python
# Feature 3: Response filtering
agent.add_response_filter("report")
response = agent.filtered_run("Generate a report on finance")
```

```
print(response)
```

### Saving and Loading State

```python
# Save the agent state
agent.save_state('saved_flow.json')

# Load the agent state
agent = Agent(llm=llm_instance, max_loops=5)
agent.load('saved_flow.json')
agent.run("Continue with the task")
```

### Async and Concurrent Execution

```python
# Run a task concurrently
response = await agent.run_concurrent("Concurrent task")
print(response)

# Run multiple tasks concurrently
tasks = [
    {"task": "Task 1"},
    {"task": "Task 2", "img": "path/to/image.jpg"},
    {"task": "Task 3", "custom_param": 42}
]
responses = agent.bulk_run(tasks)
print(responses)
```

### Various other settings

```python
# # Convert the agent object to a dictionary
print(agent.to_dict())
print(agent.to_toml())
print(agent.model_dump_json())
print(agent.model_dump_yaml())

# Ingest documents into the agent's knowledge base
agent.ingest_docs("your_pdf_path.pdf")

# Receive a message from a user and process it
agent.receive_message(name="agent_name", message="message")

# Send a message from the agent to a user
agent.send_agent_message(agent_name="agent_name", message="message")

# Ingest multiple documents into the agent's knowledge base
```

```
agent.ingest_docs("your_pdf_path.pdf", "your_csv_path.csv")

# Run the agent with a filtered system prompt
agent.filtered_run(
    "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria?"
)

# Run the agent with multiple system prompts
agent.bulk_run(
    [
        "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria?",
        "Another system prompt",
    ]
)

# Add a memory to the agent
agent.add_memory("Add a memory to the agent")

# Check the number of available tokens for the agent
agent.check_available_tokens()

# Perform token checks for the agent
agent.tokens_checks()

# Print the dashboard of the agent
agent.print_dashboard()
```

~~~
# Fetch all the documents from the doc folders
agent.get_docs_from_doc_folders()

# Activate agent ops
agent.activate_agentops()
agent.check_end_session_agentops()

# Dump the model to a JSON file
agent.model_dump_json()
print(agent.to_toml())
```
~~~