Title: Building Custom Swarms with Multiple Agents: A Comprehensive Guide for Swarm Engineers

Introduction

As artificial intelligence and machine learning continue to grow in complexity and applicability, building systems that can harness multiple agents to solve complex tasks becomes more critical. Swarm engineering enables AI agents to collaborate and solve problems autonomously in diverse fields such as finance, marketing, operations, and even creative industries. In this guide, we'll focus on how to build a custom swarm system that integrates multiple agents into a cohesive system capable of solving tasks collaboratively.

The swarm we'll design will leverage Python, use types for better code structure, and feature logging with the powerful **loguru** logging library. We'll break down how to define and initialize swarms, make them scalable, and create methods like `run(task: str)` to trigger their execution.

By the end of this article, you will have a complete understanding of:
- What swarms are and how they can be built.
- How to intake multiple agents using a flexible class.
- How to run tasks across agents and capture their outputs.
- Best practices for error handling, logging, and optimization.

---

1. Understanding the Concept of a Swarm

A **swarm** refers to a collection of agents that collaborate to solve a problem. Each agent in the swarm performs part of the task, either independently or by communicating with other agents. Swarms are ideal for:

- **Scalability**: You can add or remove agents dynamically based on the task's complexity.
- **Flexibility**: Each agent can be designed to specialize in different parts of the problem, offering modularity.
- **Autonomy**: Agents in a swarm can operate autonomously, reducing the need for constant supervision.

We'll be using Python as the primary programming language and will structure the swarm class using clean, reusable code principles.

---

2. Designing the Swarm Class: Intake Multiple Agents

We'll begin by creating a base class for our swarm. This class will intake multiple agents and define a `run` method, which is the core method for executing tasks across the swarm. Each agent is defined by its specific behavior or "intelligence" to complete part of the task.

2.1 Importing the Required Libraries and Dependencies

We'll rely on the **loguru** logging library, Pydantic for metadata handling, and standard Python typing.

```python
```

```
from typing import List, Union
from loguru import logger
from swarms.structs.base_swarm import BaseSwarm


class SwarmExecutionError(Exception):
    """Custom exception for handling swarm execution errors."""
    pass
```

2.2 Defining the Swarm Class

The class `CustomSwarm` will take in a list of agents. The agents will be instances of `BaseSwarm` (or callable functions). The `run(task: str)` method will delegate tasks to each agent in the swarm and handle any errors or retries.

```python
class CustomSwarm:
    def __init__(self, agents: List[BaseSwarm]):
        """
        Initializes the CustomSwarm with a list of agents.

        Args:
            agents (List[BaseSwarm]): A list of agent objects that inherit from BaseSwarm.
        """
        self.agents = agents
        self.validate_agents()
```

```
    def validate_agents(self):
        """Validates that each agent has a 'run' method."""
        for agent in self.agents:
            if not hasattr(agent, 'run'):
                raise AttributeError(f"Agent {agent} does not have a 'run' method.")
            logger.info(f"Agent {agent} validated successfully.")

    def run(self, task: str):
        """
        Runs the task across all agents in the swarm.

        Args:
            task (str): The task to pass to each agent.
        """
        logger.info(f"Running task '{task}' across all agents in the swarm.")
        for agent in self.agents:
            try:
                agent.run(task)
                logger.info(f"Agent {agent} successfully completed the task.")
            except Exception as e:
                logger.error(f"Agent {agent} failed to run task: {e}")
                raise SwarmExecutionError(f"Execution failed for {agent}. Task: {task}")
```

3. Adding Logging and Error Handling with `loguru`

Logging is crucial for production-grade systems, especially when managing complex tasks that involve multiple agents. **Loguru** is a simple and efficient logging library that allows us to log everything from information messages to errors.

```python
from loguru import logger

class CustomSwarm:
    def __init__(self, agents: List[BaseSwarm]):
        self.agents = agents
        logger.info("CustomSwarm initialized with agents.")
        self.validate_agents()

    def run(self, task: str):
        logger.info(f"Task received: {task}")
        for agent in self.agents:
            try:
                agent.run(task)
                logger.success(f"Agent {agent} completed task successfully.")
            except Exception as e:
                logger.error(f"Error while running task '{task}' for {agent}: {e}")
                raise SwarmExecutionError(f"Execution failed for {agent}")
```

4. Running Tasks Across Multiple Agents

The `run(task: str)` method will handle distributing the task to each agent in the swarm. Each agents `run` method is expected to take a task as input and perform its specific logic. We can add further customization by allowing each agent to return output, which can be collected for later analysis.

4.1 Example of Integrating Agents

Let's take a look at how we can define agents using the `BaseSwarm` class and integrate them into the swarm.

```python
class FinancialAgent(BaseSwarm):
    def run(self, task: str):
        logger.info(f"FinancialAgent processing task: {task}")
        # Custom logic for financial analysis
        return f"FinancialAgent response to task: {task}"

class MarketingAgent(BaseSwarm):
    def run(self, task: str):
        logger.info(f"MarketingAgent processing task: {task}")
        # Custom logic for marketing analysis
        return f"MarketingAgent response to task: {task}"
```

Now, we initialize the swarm with these agents:

```python
if __name__ == "__main__":
    agents = [FinancialAgent(), MarketingAgent()]
    swarm = CustomSwarm(agents)
    swarm.run("Analyze Q3 financial report and marketing impact.")
```

5. Enhancing the Swarm with Concurrent Execution

When dealing with large or time-consuming tasks, running agents concurrently (in parallel) can significantly improve performance. We can achieve this by utilizing Pythons **concurrent.futures** or **threading** libraries.

5.1 Running Swarms Concurrently

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

class CustomSwarm:
    def __init__(self, agents: List[BaseSwarm], max_workers: int = 4):
        self.agents = agents
        self.thread_pool = ThreadPoolExecutor(max_workers=max_workers)
        logger.info("CustomSwarm initialized with concurrent execution.")

    def run(self, task: str):
        futures = []
```

```
        for agent in self.agents:
            futures.append(self.thread_pool.submit(agent.run, task))

        for future in as_completed(futures):
            result = future.result()
            logger.info(f"Agent result: {result}")
```

6. Advanced Error Handling and Retries

In a production system, agents might fail due to a wide range of reasons (network errors, API rate limits, etc.). To ensure resilience, we can add retry mechanisms and even fallback agents that attempt to recover the failed task.

```python
class CustomSwarm:
    def run_with_retries(self, task: str, retries: int = 3):
        """
        Runs the task across all agents with retry logic.

        Args:
            task (str): The task to run.
            retries (int): Number of retries allowed for failed agents.
        """
        for agent in self.agents:
            attempt = 0
```

```
        while attempt <= retries:
            try:
                agent.run(task)
                logger.success(f"Agent {agent} completed task.")
                break
            except Exception as e:
                logger.error(f"Agent {agent} failed on attempt {attempt + 1}. Error: {e}")
                attempt += 1
                if attempt > retries:
                    logger.error(f"Agent {agent} exhausted retries. Task failed.")
```

7. Adding Documentation with Docstrings

Clear and concise documentation is critical, especially for engineers maintaining and scaling the system. Using Pythons docstrings, we can document each class and method, describing what they do and their expected inputs/outputs.

```python
class CustomSwarm:
    """
    A class to manage and execute tasks using a swarm of agents.

    Attributes:
        agents (List[BaseSwarm]): A list of agent instances.
```

```
Methods:
    run(task: str): Runs a task across all agents in the swarm.
    validate_agents(): Validates that each agent has a run method.
    run_with_retries(task: str, retries: int): Runs the task with retry logic.
"""

def __init__(self, agents: List[BaseSwarm]):
    """
    Initializes the CustomSwarm with a list of agents.

    Args:
        agents (List[BaseSwarm]): A list of agent objects that inherit from BaseSwarm.
    """
    self.agents = agents

def run(self, task: str):
    """
    Runs the task across all agents in the swarm.

    Args:
        task (str): The task to pass to each agent.
    """
    pass

def validate_agents(self):
    """Validates that each agent has a 'run' method."""
```

```
        pass
```

```

### Conclusion

Building custom swarms that intake multiple agents can drastically improve the scalability, efficiency, and flexibility of AI-driven systems. By designing a robust swarm class that manages agents, distributes tasks, and ensures error resilience, you can handle complex, multi-agent workloads efficiently.

In this blog, we've covered:
- Designing a basic swarm class.
- Running tasks across multiple agents.
- Leveraging logging, error handling, retries, and concurrency.
- Documenting your class for future-proofing.

This approach sets the foundation for building more advanced and domain-specific swarms in areas like finance, marketing, operations, and beyond. Swarm engineers can now explore more complex, multi-agent systems and push the boundaries of AI collaboration.

Stay tuned for future updates on more advanced swarm functionalities!
```