Chapter 3 : CrewAI – Framework for Multi-Agent Collaboration

Table of Contents

Table of Contents

3.1 Introduction to CrewAI

CrewAI is a conceptual framework and practical pattern for building cooperative multi-agent systems. Instead of a single large monolithic model, CrewAI structures the solution as smaller, specialized agents (researcher, analyzer, editor, publisher, etc.) that collaborate. The framework emphasizes clear role definitions, explicit task contracts, and small units of work — making systems easier to test, maintain, and scale.

3.1.1 Motivation and Purpose

- Specialization: Different agents encapsulate separate expertise (retrieval, summarization, verification), leading to more focused and capable performance on specific tasks.
- Robustness: Failure in one agent can be isolated and retried without breaking the whole pipeline, improving system resilience.
- Interpretability: Intermediate outputs create natural checkpoints for inspection and debugging, making it easier to understand the system's reasoning.
- Composability: Agents are reusable building blocks that can be easily repurposed across different workflows.

3.1.2 Core Philosophy

The core philosophy centers on a few guiding principles. Keep agents small and focused. Define clear task contracts (input schema, output schema). Prefer explicit communication channels over implicit side effects. Build using testable mock tools first, then add real connectors. This philosophy is inspired by object-oriented and microservices design, applying principles of loose coupling and high cohesion to the world of large language models (LLMs).

3.1.3 Applications Across Domains

Use cases include:

News aggregation and briefing systems.

Automated research assistants.

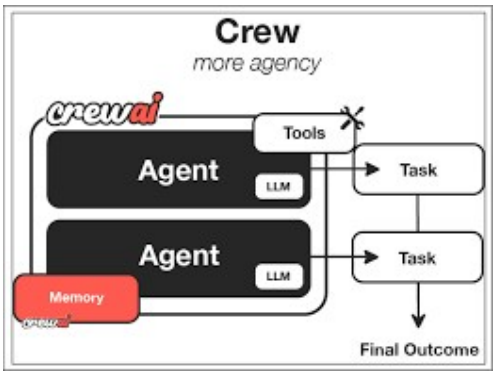
Content production pipelines (e.g., blog posts, marketing copy).

Monitoring and incident triage in operations.

Multi-robot coordination.

3.2 Core Concepts of CrewAI

The framework centers on a set of core abstractions that define how agents are built and coordinated.



3.2.1 Agent

An Agent is an autonomous unit with a defined role, a goal, optional tools, and an act() method that produces outputs from inputs. The agent is essentially the worker in the system, often backed by an LLM, responsible for carrying out specific parts of the workflow.

3.2.2 Task

A Task is a unit of work assigned to an agent. Tasks include a clear description, necessary input data, and expected outputs. Defining precise tasks is key to receiving high-quality results. A task acts as a detailed prompt wrapper, providing the agent with the necessary context and constraints to execute a specific, measurable unit of work.

3.2.3 Crew

A Crew is a composed set of agents and a process definition that orchestrates task execution. It serves as the manager and execution environment for the collaborative workflow, ensuring agents work together coherently.

3.2.4 Process

Process defines the execution strategy for the tasks within the Crew. The primary strategies are: SEQUENTIAL, PARALLEL, or HIERARCHICAL.

3.2.5 Memory and Context

Memory stores context across tasks and runs, enabling agents to refer to previous steps or shared information. It may be a simple key-value store or backed by a vector database for richer context retrieval (RAG).

3.2.6 Delegation

Delegation lets agents create subtasks and assign them to others, improving flexibility for handling complex or unexpected sub-problems during execution, essentially enabling a manager-worker dynamic.

3.3 Installation & Environment Setup

3.3.1 Prerequisites

Python 3.9+ is recommended. Always use virtual environments (venv) or conda to maintain project-specific dependencies.

3.3.2 Installing dependencies

```
python -m venv venv

source venv/bin/activate # Linux / macOS

.\venv\Scripts\activate # Windows PowerShell

pip install requests rich

Optional for real LLM: pip install openai
```

3.3.3 Project layout

A typical project structure for a CrewAI application:

crewai_chapter3/

├── crewai_core.py # minimal framework (provided below)

├── tools.py # web / calendar mock tools

├── content_case.py # example crew and case study

├── README.md

└── requirements.txt

3.3.4 Verification

Quick import check to verify the environment setup:

```
python -c "import crewai_core; print('OK')"
```

3.4 Defining and Customizing Agents

3.4.1 Agent roles & goals

Roles should be concise (e.g., Researcher, Writer, Editor, Publisher). Goals define the agent’s overall objective in one clear sentence (e.g., "To find and summarize the top three most influential AI trends of the week").

3.4.2 Backstories & behavior tuning

Backstories are short, persona-driven prompts that guide the agent's writing style, tone, and assumptions. This is a critical component of prompt engineering for shaping agent behavior, making them sound professional, witty, or cautious, as needed.

3.4.3 Tools and capabilities

Tools are Python callables (functions or classes) attached to agents, enabling them to interact with the external world (e.g., search, file systems, APIs). Example tool signature:

```
def web_search(query: str) -> list:

return ['result1', 'result2']
```

3.4.4 Inter-agent communication patterns

Common communication patterns include shared memory (simplest), message queues (for asynchronous communication), or direct callbacks/function calls between agents. The choice depends on the complexity and reliability requirements of the crew.

3.5 Managing Tasks & Workflows

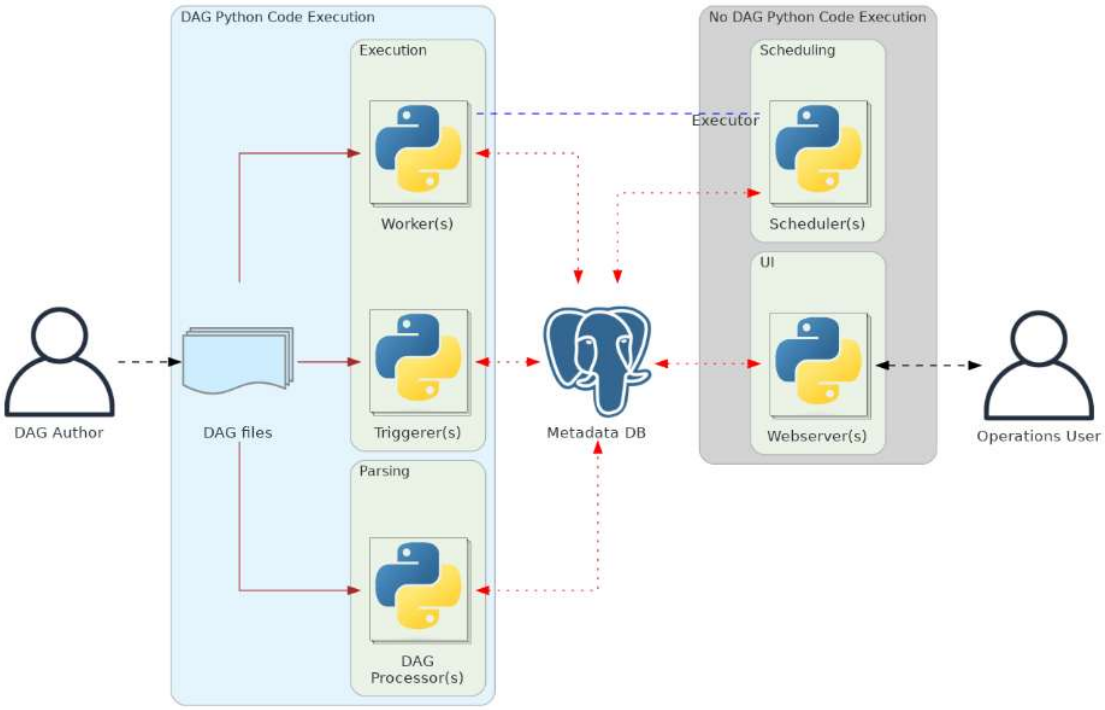
3.5.1 Sequential workflows

Sequential pipelines are deterministic, running tasks one after the other. The output of one task often becomes the input or context for the next, forming a clear chain of responsibility.

```
crew.add_task(Task("collect","collect trend data", researcher))

crew.add_task(Task("write","write article", writer))

crew.kickoff()
```



3.5.2 Parallel workflows

Parallel is used when tasks are independent and can be executed simultaneously (e.g., fetching news, weather, and stock data). This significantly reduces total execution time by leveraging concurrent processing.

3.5.3 Hierarchical workflows (manager pattern)

A manager agent is responsible for breaking down a large problem, spawning subtasks, monitoring their execution by worker agents, and aggregating the final result. This pattern helps tackle complex aggregation and decision-making tasks efficiently.

3.5.4 Monitoring, logging, retries

It is essential to use structured logs for easy analysis and to add retry loops for external calls (like API access) to handle transient failures. Logs should be persisted for auditing and performance analysis to diagnose bottlenecks.

3.6 Integrating Tools and External APIs

3.6.1 Web search & scraping

For mock mode development, use canned responses. For production, use proper search APIs (like Google Search API) and ensure respectful scraping practices (adhering to robots.txt) to access real-time external data.

tools.py - mock search

```
def mock_search(query):

samples = {

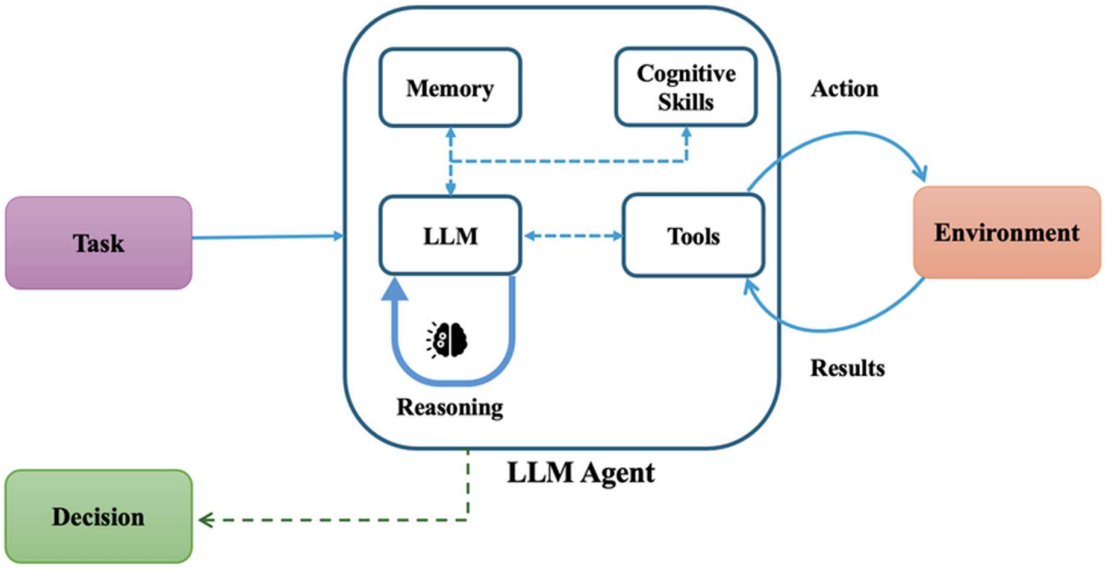
"ai trends": ["Trend A: LLMs get better", "Trend B: Retrieval-augmented
     generation"]

}

return samples.get(query.lower(), ["No results found"])
```

3.6.2 Calendar, email, and storage APIs

Integrating productivity tools requires proper authentication. For Google Calendar, this means OAuth; for email, use SMTP or services like SendGrid; for cloud storage, use S3/GCS SDKs. Agents use these tools to perform real-world actions.



3.6.3 Secrets & API keys management

Never hardcode sensitive information. Store keys in environment variables or dedicated secret managers (e.g., Vault, AWS Secrets Manager). DO NOT commit them to source control.

3.6.4 Tool chaining and caching

Tool chaining involves an agent using the output of one tool as the input for another (e.g., search -> retrieve document -> summarize). Caching intermediate results reduces costs and speeds up execution, particularly for expensive LLM calls or API lookups.

3.7 Building a Multi-Agent Crew System

3.7.1 Define the team & ownership

Map domain responsibilities to agents, ensuring no overlap or ambiguity. For reliability, create unit tests for each agent's success criteria and integration tests for the full workflow.

3.7.2 Crew orchestration example

```
Example usage with minimal framework:

crew = Crew(agents=[researcher, writer, editor], process="SEQUENTIAL")

crew.add_task(Task("collect","Collect trend data", researcher))

crew.add_task(Task("write","Write article", writer))

res = crew.kickoff()
```

3.7.3 Runtime architecture and deployment

For large-scale systems, consider running agents as microservices for isolation. Use message queues (like RabbitMQ or Kafka) for reliable communication, and externalize memory (e.g., Redis, database) to make agents stateless and horizontally scalable.

3.8 Case Study: AI Content Creator Crew

3.8.1 Overview

Goal: produce a daily short blog article on trending AI topics using a small editorial crew of agents. This demonstrates the sequential workflow and memory sharing capabilities.

3.8.2 Agent definitions

Researcher: Collects headlines and insights (uses search tool).

Writer: Drafts the article based on the Researcher's output.

Editor: Refines the draft for style, grammar, and conciseness.

Publisher: Finalizes and "publishes" the content (in a real system, this would involve an API call to a CMS).

3.8.3 Code: full working example (mock + real mode)

```
# crewai_core.py (minimal framework)

```python

import threading

from typing import Callable, Any, Dict, List, Optional

from queue import Queue, Empty


class Task:

    def __init__(self, name: str, description: str, agent: 'Agent',
     input_data: Any = None):

        self.name = name

        self.description = description

        self.agent = agent

        self.input_data = input_data

        self.output = None

        self.status = 'PENDING'
```
```

```
class Agent:

    def __init__(self, role: str, goal: str, backstory: str = "", tools:
     Optional[Dict[str, Callable]] = None):

        self.role = role

        self.goal = goal

        self.backstory = backstory

        self.tools = tools or {}



    def act(self, task: Task, memory: Dict[str, Any]) -> Any:

        # Placeholder for LLM interaction logic

        return {"role": self.role, "task": task.name, "result":
     f"Processed {task.description}"}




class Crew:

    def __init__(self, agents: List[Agent], process: str = "SEQUENTIAL",
     verbose: bool = True):

        self.agents = agents

        self.process = process

        self.verbose = verbose

        self.memory = {} # Simple in-memory context store

        self.tasks: List[Task] = []



    def add_task(self, task: Task):

        self.tasks.append(task)



    def kickoff(self, timeout_per_task: int = 30):

        if self.process == "SEQUENTIAL":

            return self._run_sequential(timeout_per_task)

        elif self.process == "PARALLEL":
```

```
        return self._run_parallel(timeout_per_task)
    else:
        raise ValueError("Unknown process")



def _run_sequential(self, timeout_per_task):
    results = {}
    for task in self.tasks:
        if self.verbose:
            print(f"[Crew] Running task: {task.name} -> agent:
 {task.agent.role}")
        task.status = 'RUNNING'
        # The agent uses its act method, potentially looking at
 self.memory
        out = task.agent.act(task, self.memory)
        task.output = out
        task.status = 'COMPLETED'
        results[task.name] = out
        self.memory[task.name] = out # Store result in memory for
 subsequent agents
    return results



def _run_parallel(self, timeout_per_task):
    q = Queue()
    results = {}


    def worker(task: Task):
        if self.verbose:
            print(f"[Crew] (parallel) Worker starting task:
 {task.name}")
```

```
            task.output = task.agent.act(task, self.memory) # Note:
        Parallel memory access needs care in real systems
            task.status = 'COMPLETED'
            q.put((task.name, task.output))



        threads = []
        for t in self.tasks:
            th = threading.Thread(target=worker, args=(t,))
            th.start()
            threads.append(th)



        for th in threads:
            th.join(timeout=timeout_per_task)

        while True:
            try:
                name, out = q.get_nowait()
                results[name] = out
                self.memory[name] = out
            except Empty:
                break
        return results
```

content_case.py (mock pipeline)

```
Python
from crewai_core import Agent, Task, Crew



class ResearcherAgent(Agent):
```

```
    # Overrides base Agent's act() for specific logic
    def act(self, task, memory):
        query = task.input_data or "ai trends"
        # Access the 'search' tool defined in the constructor
        results = self.tools.get('search', lambda q: ["nodata"])(query)
        return {"query": query, "hits": results}

class WriterAgent(Agent):
    def act(self, task, memory):
        # Access the result of the 'collect' task from memory
        research = memory.get('collect')
        if not research:
            return {"error": "no research data available from previous
     task"}

        content = "Article Title: Trends in AI\n\n"
        for i, h in enumerate(research['hits'], 1):
            content += f"{i}. {h}\n\n"
        return {"article": content}

class EditorAgent(Agent):
    def act(self, task, memory):
        # Access the article content from memory
        article = memory.get('write', {}).get('article', '')

        # Simple mock editing: remove double newlines
        edited = article.replace("\n\n", "\n")
        return {"edited_article": edited}

# Mock tool definition
```

```
def mock_search(q):
    return [f"Mocked result for '{q}' - insight{i}" for i in range(1,4)]



# 1. Instantiate Agents with roles, goals, and tools
researcher = ResearcherAgent("Researcher","Find trends", tools={'search':
    mock_search})
writer = WriterAgent("Writer","Compose article")
editor = EditorAgent("Editor","Edit article")


# 2. Instantiate Crew with agents and process type
crew = Crew(agents=[researcher, writer, editor], process="SEQUENTIAL",
    verbose=True)


# 3. Define and add Tasks
crew.add_task(Task("collect","Collect trend data", researcher,
    input_data="ai trends"))
crew.add_task(Task("write","Write article", writer))
crew.add_task(Task("edit","Edit article", editor))


# 4. Kickoff the execution
if __name__ == '__main__':
    res = crew.kickoff()
    print('\n== Final Results ==')
    for k, v in res.items():
        print(f"\n--- Task: {k} ---")
        print(v)
```

3.8.4 Sample run and expected output

Run python content_case.py and expect printed results for each task. The final memory dictionary will contain keys collect, write, and edit with their respective outputs, demonstrating the sequential passing of context.

3.9 Best Practices & Optimization

3.9.1 Prompt engineering

This is the most direct way to tune agent quality. Keep instructions explicit. Provide examples and constraints on output format (e.g., JSON schema) to minimize generation errors. Use clear role-play and high-quality "backstories."

3.9.2 Memory usage & pruning

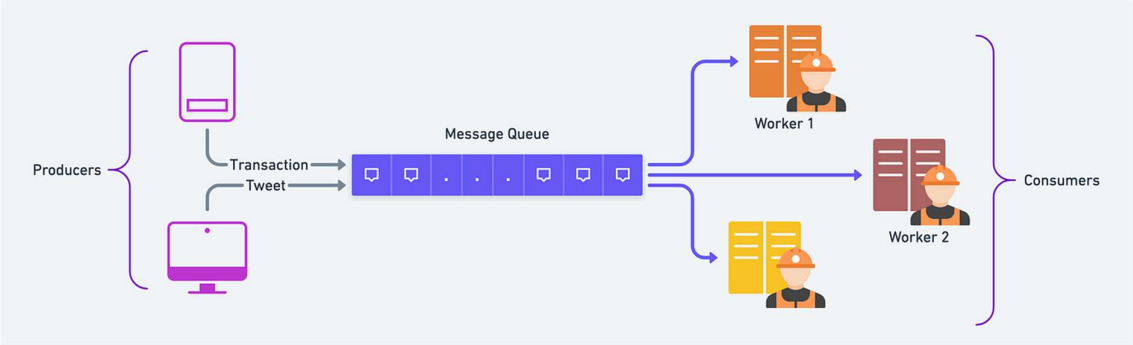
Raw transcripts quickly exceed LLM context windows and increase cost. Store summaries rather than raw transcripts, prune old entries, and use Time-To-Live (TTL) for long-term stores to manage memory efficiently.

3.9.3 Observability

Log structured JSON for each task run to track input, output, and status. Collect metrics for latency and success rates. Crucially, track token usage in LLM systems to monitor costs and optimize efficiency.

3.9.4 Scaling patterns

To handle high load, make agents stateless where possible. Externalize memory (see 3.9.2) and employ worker pools and message queues for asynchronous, distributed execution.



3.10 Future Directions & Research

3.10.1 Distributed crews

Multi-node crews, where agents run on different machines or even different cloud providers, enable resource isolation and significantly better scaling for massive, long-running tasks.

3.10.2 Adaptive learning

Research is focused on agents that can learn delegation policies and task management strategies based on past success/failure rates, allowing the crew to improve its efficiency over time without explicit code changes.

3.10.3 Interoperability & standards

Standardizing agent interfaces and tool specifications (like the Open-Tool-Calling standard) allows for mixing and matching agents from different frameworks (e.g., CrewAI, Autogen, LangGraph) in a single, cohesive system.

3.11 Summary, Glossary, References

3.11.1 Key takeaways

- CrewAI emphasizes **modular, testable** multi-agent systems built around clear roles.
- Start development with **mock tools** and progressively integrate real connectors.
- **Observability** (structured logging) and effective **memory management** are critical for production viability.

Glossary (abridged)

- **Agent:** Autonomous worker that performs tasks using tools and an LLM.
- **Task:** A unit of work with a description, input, and expected output.
- **Crew:** Collection of agents plus the orchestration logic (Process).
- **Process:** The execution strategy (Sequential, Parallel, Hierarchical).
- **Memory:** Persistent context store that shares information between tasks.

References

- CrewAI (example project): https://github.com/joaomdmoura/crewAI
- Autogen multi-agent approach: https://github.com/microsoft/autogen
- LangGraph: https://github.com/langchain-ai/langgraph