Chapter 6: CrewAI – Framework for Multi-Agent Collaboration

Course Module: Understanding and Implementing CrewAI

Table of Contents

Contents

6.1 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.

6.1.1 Motivation and Purpose

Specialization: different agents encapsulate separate expertise (retrieval, summarization, verification).

Robustness: failure in one agent can be isolated and retried without breaking the whole pipeline.

Interpretability: intermediate outputs create natural checkpoints for inspection.

Composability: agents are reusable building blocks.

6.1.2 Core Philosophy

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.

6.1.3 Applications Across Domains

Use cases include:

- News aggregation and briefing systems.
- Automated research assistants.
- Content production pipelines.
- Monitoring and incident triage in operations.
- Multi-robot coordination.

6.2 Core Concepts of CrewAI

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

6.2.1 Agent

An **Agent** is an autonomous unit with a role, goal, optional tools, and an act() method that produces outputs from inputs.

6.2.2 Task

A **Task** is a unit of work assigned to an agent. Tasks include description, input data, and expected outputs.

6.2.3 Crew

A **Crew** is a composed set of agents and a process definition that orchestrates task execution.

6.2.4 Process

Process defines the execution strategy: SEQUENTIAL, PARALLEL, or HIERARCHICAL.

6.2.5 Memory and Context

Memory stores context across tasks and runs. It may be a simple key-value store or backed by a vector database.

6.2.6 Delegation

Delegation lets agents create subtasks and assign them to others, improving flexibility for complex tasks.

6.3 Installation & Environment Setup

6.3.1 Prerequisites

Python 3.9+ is recommended. Use virtual environments (venv) or conda to maintain dependencies.

6.3.2 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
```

6.3.3 Project layout

```
crewai_chapter6/

 ├── 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
```

6.3.4 Verification

```
Quick import check:

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

6.4 Defining and Customizing Agents

6.4.1 Agent roles & goals

Roles should be concise: e.g., Researcher, Writer, Editor, Publisher. Goals define the agent's objective in one sentence.

6.4.2 Backstories & behavior tuning

Backstories are short prompts guiding agent writing style and assumptions.

6.4.3 Tools and capabilities

Tools are callables attached to agents. Example tool signature:

```
def web_search(query: str) -> list:
   return ['result1', 'result2']
```

6.4.4 Inter-agent communication patterns

Common patterns: shared memory, message queues, or direct callbacks. Shared memory is simplest to start with.

6.5 Managing Tasks & Workflows

6.5.1 Sequential workflows

```
Sequential pipelines are deterministic:

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

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

crew.kickoff()
```

6.5.2 Parallel workflows

Parallel is used when tasks are independent (news & weather fetches).

6.5.3 Hierarchical workflows (manager pattern)

A manager agent spawns and monitors subtasks. This pattern helps complex aggregation tasks.

6.5.4 Monitoring, logging, retries

Use structured logs and add retry loops for external calls. Persist logs for auditing.

6.6 Integrating Tools and External APIs

6.6.1 Web search & scraping

For mock mode use canned responses. For production use proper APIs and respectful scraping practices.

```
# 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"])
```

6.6.2 Calendar, email, and storage APIs

Google Calendar requires OAuth; for email use SMTP or services like SendGrid; for storage use S3/GCS.

6.6.3 Secrets & API keys management

Store keys in environment variables or secret managers (DO NOT commit them).

6.6.4 Tool chaining and caching

Chain tools for multi-step retrieval and summarize intermediate results; cache to reduce costs.

6.7 Building a Multi-Agent Crew System

6.7.1 Define the team & ownership

Map domain responsibilities to agents and create tests for each agent's success criteria.

6.7.2 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()
```

6.7.3 Runtime architecture and deployment

Run agents as microservices, use message queues for reliability, and externalize memory for scaling.

6.8 Case Study: AI Content Creator Crew

6.8.1 Overview

Goal: produce a daily short blog article on trending AI topics using a small editorial crew of agents.

6.8.2 Agent definitions

Researcher collects headlines; Writer drafts; Editor refines; Publisher publishes.

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

Below are the key files. Save them and run locally.

```
# crewai_core.py (minimal framework)

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:
        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 = {}
        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'
            out = task.agent.act(task, self.memory)
            task.output = out
            task.status = 'COMPLETED'
            results[task.name] = out
            self.memory[task.name] = out
        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)
            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)
from crewai_core import Agent, Task, Crew

class ResearcherAgent(Agent):
    def act(self, task, memory):
        query = task.input_data or "ai trends"
        results = self.tools.get('search', lambda q: ["no data"])(query)
        return {"query": query, "hits": results}

class WriterAgent(Agent):
    def act(self, task, memory):
        research = memory.get('collect')
        if not research:
            return {"error": "no research"}
        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):
        article = memory.get('write', {}).get('article', '')
        edited = article.replace("\n\n", "\n")
        return {"edited_article": edited}


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


researcher = ResearcherAgent("Researcher","Find trends", tools={'search': mock_search})
writer = WriterAgent("Writer","Compose article")
editor = EditorAgent("Editor","Edit article")


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


if __name__ == '__main__':
    res = crew.kickoff()
    print('== Results ==')
    for k,v in res.items():
        print(k, '->', v)
```

6.8.4 Sample run and expected output

Run python content_case.py and expect printed results; the final memory will contain keys collect, write, and edit with their outputs.

6.9 Best Practices & Optimization

6.9.1 Prompt engineering

Keep instructions explicit. Provide examples and constraints.

6.9.2 Memory usage & pruning

Store summaries rather than raw transcripts; prune old entries; use TTL for long-term stores.

6.9.3 Observability

Log structured JSON for each task run; collect metrics for latency and success rates; track token usage in LLM systems.

6.9.4 Scaling patterns

Make agents stateless where possible; externalize memory; use worker pools and message queues.

6.10 Future Directions & Research

6.10.1 Distributed crews

Multi-node crews enable resource isolation and better scaling.

6.10.2 Adaptive learning

Agents that learn delegation policies can improve system efficiency over time.

6.10.3 Interoperability & standards

Standardizing agent interfaces and tool specs allows mixing and matching frameworks.

6.11 Summary, Glossary, References

6.11.1 Key takeaways

- CrewAI emphasizes modular, testable multi-agent systems.
- Start with mock tools and progressively integrate real connectors.
- Observability and memory management are critical.

Glossary (abridged)

Agent — autonomous worker that performs tasks.

Task — unit of work.

Crew — collection of agents plus orchestration.

Process — execution strategy.

Memory — persistent context store.

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