CrewAI Framework

Рольова модель мультиагентних систем

Від Sequential процесу до Memory-enabled Crew.

Покрокові пояснення, інструменти, приклади коду.

Модуль 5 курсу agents.pro

agents.pro / module5 - Навчальний курс

2025

1. Вступ

Цей документ є навчальним посібником до модуля agents_v2, що демонструє побудову мультиагентних систем за допомогою CrewAI - фреймворку, заснованого на рольовій моделі взаємодії агентів.

Що таке CrewAI?

CrewAI (версія 1.4.0+) - це Python-фреймворк для створення команд AI-агентів, де кожен агент має визначену роль, мету та історію. Ключова метафора - це 'екіпаж' (crew), де агенти працюють разом для досягнення спільної мети.

Основні переваги CrewAI:
- Рольова модель: агенти визначаються через role/goal/backstory
- Два типи процесів: Sequential та Hierarchical
- Параметризація задач через {variable} placeholders
- Вбудована система памяті (memory)
- Інтеграція з LangChain tools та власна бібліотека crewai_tools

Відмінність від LangChain/LangGraph

Аспект	LangChain/LangGraph	CrewAI
Підхід	Граф станів	Рольова модель
Оркестрація	StateGraph + edges	Crew + Process
Агенти	Функції-ноди	Role/Goal/Backstory
Координація	Routing functions	Manager або Sequential
Складність	Більше контролю	Простіший API
Налаштування	Гранулярне	Декларативне

2. Ключові концепції

Agent - Агент

Agent - це основна одиниця CrewAI. Кожен агент визначається через:
- role: роль агента (напр., 'Senior Research Analyst')
- goal: мета, яку агент намагається досягти
- backstory: контекст та досвід агента (покращує якість відповідей)
- llm: модель для використання (напр., 'gpt-4o-mini')
- tools: список інструментів доступних агенту
- allow_delegation: чи може агент делегувати задачі іншим

Приклад створення агента:

```
researcher = Agent(
    role="Senior Research Analyst",
    goal="Uncover cutting-edge developments about {topic}",
    backstory="You are a seasoned research analyst with 10 years...",
    verbose=True,
    allow_delegation=False,
    llm="gpt-4o-mini"
)
```

Task - Задача

Task визначає конкретну роботу для агента:
- description: що потрібно зробити (підтримує {variable} placeholders)
- expected_output: очікуваний результат
- agent: відповідальний агент

Приклад створення задачі:

```
research_task = Task(
    description="Conduct comprehensive research on {topic}...",
    expected_output="A detailed research report with 10-15 bullet points...",
    agent=researcher
)
```

Crew - Екіпаж

Crew об'єднує агентів та задачі. Конфігурується через:
- agents: список агентів
- tasks: список задач (порядок важливий для sequential)
- process: Process.sequential або Process.hierarchical
- verbose: детальне логування
- memory: включення системи памяті

Kickoff - Запуск

Метод kickoff() запускає виконання crew. Параметри передаються через inputs dict і підставляються в {variable} placeholders:

Створення та запуск crew:

```
crew = Crew(
    agents=[researcher, writer, editor],
    tasks=[research_task, writing_task, editing_task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff(inputs={
    "topic": "Multi-Agent AI Systems with LangChain and CrewAI"
})
```

3. Приклад 01: Базовий Crew

Файл: agents_v2/01_basic_crew.py

Перший приклад демонструє Sequential процес - найпростішу модель оркестрації, де задачі виконуються послідовно.

Структура: Researcher -> Writer -> Editor

Три агенти працюють послідовно над створенням контенту:

1. Senior Research Analyst - досліджує тему
2. Tech Content Writer - пише статтю на основі дослідження
3. Senior Content Editor - редагує та полірує результат

Sequential Process

Researcher -> Результат дослідження
-> Writer -> Чернетка статті
-> Editor -> Фінальний текст

Параметризація через {topic}

Всі задачі використовують {topic} placeholder. При виклику kickoff() передається конкретне значення, яке підставляється в усі описи задач та цілі агентів:

```
# В описі задачі:
description="Conduct comprehensive research on {topic}..."

# При запуску:
result = crew.kickoff(inputs={
    "topic": "Multi-Agent AI Systems"
})
```

Ключові патерни

- allow_delegation=False - кожен агент виконує тільки свою задачу
- verbose=True - детальне логування для відлагодження
- Вихід кожної задачі автоматично передається наступній
- Crew можна перевикористати з іншим topic через kickoff()

4. Приклад 02: Ієрархічний Crew

Файл: agents_v2/02_hierarchical_crew.py

Цей приклад демонструє Hierarchical процес з автоматично створеним менеджером, який координує команду з 6 спеціалістів.

Hierarchical Process

В ієрархічному режимі CrewAI автоматично створює Manager-агента, який:

1. Планує стратегію виконання
2. Делегує задачі відповідним спеціалістам
3. Перевіряє якість результатів
4. Координує передачу даних між агентами

Ієрархічна структура

MANAGER (auto-created)
|-- Requirements Analyst
|-- Software Architect
|-- Backend Developer
|-- Frontend Developer
|-- QA Engineer
|-- Documentation Specialist

6 спеціалізованих агентів

Агент	Роль	Інструменти
Requirements Analyst	Аналіз вимог	-
Software Architect	Дизайн архітектури	-
Backend Developer	Backend розробка	-
Frontend Developer	Frontend розробка	-
QA Engineer	Тестування	-
Documentation Spec.	Збереження у файл	FileWriterTool

FileWriterTool

Documentation Specialist використовує FileWriterTool для збереження результатів у файл project_deliverables.md:

```
from crewai_tools import FileWriterTool

documentation_specialist = Agent(
    role="Documentation Specialist",
    goal="Compile and save all project deliverables...",
    tools=[FileWriterTool()],
    llm="gpt-4o-mini"
)
```

Конфігурація manager_llm

Параметр manager_llm обов'язковий для ієрархічного процесу. Manager не входить в список agents - він створюється автоматично:

```
crew = Crew(
    agents=[analyst, architect, backend_dev, frontend_dev, qa, docs],
    tasks=[req_task, arch_task, back_task, front_task, test_task, save_task],
    process=Process.hierarchical,
    verbose=True,
    manager_llm="gpt-4o-mini"
)
```

5. Приклад 03: Crew з інструментами

Файл: agents_v2/03_research_crew_with_tools.py

Цей приклад показує інтеграцію різних типів інструментів: custom tools через @tool декоратор, crewai_tools бібліотеку та LangChain tools.

Типи інструментів

1. Custom tools (@tool декоратор)

Власні функції, обгорнуті декоратором @tool з langchain_core.tools. Кожен tool має назву, опис та параметри:

```
from langchain_core.tools import tool

@tool
def analyze_data(data_json: str) -> str:
    """Analyze JSON data and return statistical insights."""
    data = json.loads(data_json)
    if isinstance(data, list):
        return f"Dataset contains {len(data)} items."
    return "Data analyzed successfully"

@tool
def calculate_metrics(expression: str) -> str:
    """Safely evaluate mathematical expressions."""
    allowed_names = {"abs": abs, "min": min, "max": max, "sum": sum}
    result = eval(expression, {"__builtins__": {}}, allowed_names)
    return f"Result: {result}"
```

2. crewai_tools бібліотека

Готові інструменти:

```
from crewai_tools import FileReadTool, DirectoryReadTool

file_read_tool = FileReadTool()
directory_read_tool = DirectoryReadTool()
```

3. LangChain tools

Пошук через DuckDuckGo:

```
from langchain_community.tools import DuckDuckGoSearchResults

search_tool = DuckDuckGoSearchResults(num_results=5)
```

Призначення tools агентам

Інструменти призначаються конкретним агентам через параметр tools. Кожен агент отримує тільки ті інструменти, які потрібні для його ролі:

Агент	Tools	Призначення
Researcher	FileRead, DirRead, DDG	Збір інформації
Analyst	analyze_data, calculate	Аналіз даних
Writer	(немає)	Написання звіту

Множинна параметризація

На відміну від Приклад 01, тут використовуються кілька параметрів:

```
inputs = {
    "research_topic": "Multi-Agent AI Frameworks Comparison",
    "focus_areas": "architecture, ease of use, tool integration",
    "target_audience": "Technical leaders and AI engineers"
}
result = crew.kickoff(inputs=inputs)
```

6. Приклад 04: Memory-enabled Crew

Файл: agents_v2/04_memory_enabled_crew.py

Цей приклад демонструє систему памяті CrewAI - здатність зберігати контекст між викликами kickoff() для персоналізованих відповідей.

Увімкнення памяті

Конфігурація memory:

```
crew = Crew(
    agents=[context_analyzer, knowledge_curator, assistant],
    tasks=[context_task, knowledge_task, response_task],
    process=Process.sequential,
    verbose=True,
    memory=True,              # Увімкнути пам'ять
    embedder={
        "provider": "openai",
        "config": {
            "model": "text-embedding-3-small"
        }
    }
)
```

4 типи памяті

Тип	Опис	Збереження
Short-term	Контекст поточної розмови	В сесії
Long-term	Довготривала інформація	Між сесіями
Entity	Дані про конкретні сутності	Між сесіями
Contextual	Контекст задач та агентів	В сесії

Multi-turn conversations

Приклад симулює багатокрокову розмову, де кожен наступний виклик kickoff() враховує попередній контекст:

```
# Turn 1: Користувач ділиться контекстом
result1 = crew.kickoff(inputs={
    "user_request": "I'm working on a multi-agent AI project using LangChain...",
    "tone": "professional and helpful"
})

# Turn 2: Crew пам'ятає про LangChain проект
result2 = crew.kickoff(inputs={
    "user_request": "What's the best way to structure my agent code?",
    "tone": "professional and helpful"
})

# Turn 3: Crew пам'ятає весь контекст
result3 = crew.kickoff(inputs={
    "user_request": "Can you recommend testing strategies for this?",
    "tone": "professional and helpful"
})
```

3 агенти для роботи з пам'яттю

- Context Analyzer - аналізує запит в контексті попередніх розмов
- Knowledge Curator - організовує та витягує релевантну інформацію з памяті
- Personal Assistant - створює персоналізовану відповідь

Embedder конфігурація

Для роботи памяті потрібен embedder. CrewAI підтримує OpenAI, Cohere та HuggingFace embeddings. Рекомендовано text-embedding-3-small від OpenAI.

7. Sequential vs Hierarchical

Порівняння процесів

Аспект	Sequential	Hierarchical
Потік	Лінійний A->B->C	Manager делегує
Гнучкість	Низька	Висока
Передбачуваність	Висока	Середня
Вартість (tokens)	Нижча	Вища (+manager)
Складність debug	Проста	Складніша
Кількість агентів	2-4 оптимально	5+ оптимально

Коли використовувати Sequential?

- Прості лінійні workflows (дослідження -> написання -> редагування)
- Передбачувана послідовність кроків
- Невелика кількість агентів (2-4)
- Обмежений бюджет на API виклики
- Потрібна простота відлагодження

Коли використовувати Hierarchical?

- Складні проекти з багатьма спеціалістами (5+)
- Потрібна валідація якості на кожному етапі
- Динамічне визначення порядку виконання
- Можливість перепризначення задач при невдачі
- Великі команди з чітким розподілом відповідальності

8. Налаштування та запуск

Швидкий старт

```
cd agents_v2
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt
cp .env.example .env  # заповніть API ключі
python 01_basic_crew.py
```

Необхідні змінні оточення

Змінна	Обов'язкова	Джерело
OPENAI_API_KEY	Так	platform.openai.com
CREWAI_TELEMETRY	Ні (false)	Вимкнути телеметрію

Ключові залежності

requirements.txt (основні):

```
crewai>=1.4.0
crewai-tools>=0.38
langmem>=0.0.20
langchain>=1.0
duckduckgo-search
```

Python версія

Рекомендовано Python 3.10-3.13. Уникайте Python 3.14 через проблеми сумісності з Pydantic.

Важливі зауваження

- Кожен скрипт запускається окремо (не пакет)
- Скрипти використовують input() для пауз між тестами
- verbose=True рекомендовано для навчання і відлагодження
- Telemetry CrewAI краще вимкнути: CREWAI_TELEMETRY=false
- FileWriterTool в 02_* створює файли в поточній директорії