# CrewAI Review

**CrewAI** is an advanced multi-agent orchestration framework designed to enable multiple AI “personas” to collaborate toward complex goals. Built in Python as an independent, open-source project, CrewAI provides an architecture that combines specialized agent roles, configurable workflows, and integration with external tools, making it a powerful resource for developers and technical teams looking to automate processes and build intelligent solutions.

It goes beyond executing isolated tasks — CrewAI allows agents to communicate with one another, share intermediate results, divide responsibilities, use iterative reasoning, and make coordinated decisions to deliver high-quality outputs. It can be applied to scenarios such as research, content generation, software development, customer support, and data analysis, functioning as a coordinated “crew” of AI agents that turn human intent into executable actions.

## How it works

CrewAI operates based on **agents** and **tasks** defined by the user, typically through natural language prompts or Python scripts. Each agent is assigned a role (e.g., researcher, writer, analyst), a set of objectives, and an allowed set of tools or APIs.

When a process begins, the system distributes tasks among the agents, who collaborate sequentially or in parallel, exchanging information and adjusting their strategies based on partial results. This coordination is managed by an internal *process manager*, which ensures task dependencies are respected and the execution follows the intended flow.

Recent versions of CrewAI include support for **code execution within agents**, integration with local models via Ollama or LM Studio, and the ability to combine creative, autonomous workflows (“Crews”) with deterministic, auditable pipelines (“Flows”), giving developers more flexibility and fine-grained control.

## User interface and workflow

In its current form, CrewAI is primarily code-driven, though it offers examples and templates that make creating agent crews easier. The typical workflow involves:

1. **Defining agents** — including their roles, goals, and tools.
2. **Configuring tasks** — specifying what needs to be done, along with dependencies.
3. **Choosing execution style** — sequential, parallel, or hierarchical.
4. **Running the crew** — agents interact, use tools, and share information to produce results.
5. **Reviewing outputs** — results can be programmatically validated, transformed, or passed to the next stage in a pipeline.

## Performance Considerations

One trade-off with CrewAI's flexibility is the need to manage agent complexity — particularly when using local LLMs, such as those I ran through LM Studio in my work. Running multiple agents in parallel can be resource-intensive, consuming significant CPU/GPU power and memory, and increasing token usage.

In my setup, this hardware dependency caused inconsistent performance: sometimes the pipeline ran smoothly, while other times agents stalled or produced incomplete results. Additionally, the model occasionally hallucinated documentation for non-existent files or ignored tool outputs entirely. As the number of agents and logical steps grew, these issues became more frequent, particularly near token limits.

Careful design of agent roles, task scope, and tool permissions is essential to minimize these problems. Limiting unnecessary tool calls, optimizing prompts for clarity, and reducing the number of simultaneous agents can improve stability.

Despite these challenges, CrewAI's architecture remains particularly valuable for multi-stage, high-context workflows — such as documentation pipelines — where splitting the goal into specialized subtasks (file reading, code analysis, documentation writing) and letting different agents handle each phase produces more structured, consistent results than a single-agent approach.

## How I Used CrewAI

I used **CrewAI** to automate the documentation of Python code using a local LLM through **LM Studio**, since I did not have an OpenAI API key. This allowed me to process and generate documentation entirely offline, giving me more control and privacy during analysis.

The workflow included:

- Reading source files from various inputs: direct raw GitHub URLs, cloned repositories, and local files.
- Analyzing code structure to identify classes, functions, and logic flow.
- Generating technical documentation in natural language.

The process could be triggered manually or automatically via a GitHub webhook. For local development, I exposed the webhook server with ngrok so GitHub could send events directly to my machine.

Agents worked in sequence across two crews (Download & Extract → Documentation). Depending on the trigger (raw_link vs clone_repo), different agents were activated:

***Download & Extract Crew***

- **GitHub File Downloader** *(raw_link only)* — fetched a single raw file from GitHub.
- **Repo Cloner** *(clone_repo only)* — cloned the target repository.
- **File Lister** — enumerated and filtered files using directory rules.
- **File Content Reader** — read file contents for downstream analysis.

- **Code Splitter Agent** — split large code files into manageable chunks.

***Documentation Crew***

- **Raw Code Reader** *(raw_link)* / **Code Reader** *(clone_repo)* — loaded the code to be analyzed.
- **Code Insight Agent** — analyzed structure, symbols, and logic flow.
- **Doc Writer** — generated human-readable technical documentation.
- **Markdown Formatter** — normalized and saved the final output (documentacao_final.md).

While this approach was highly effective for structured documentation, performance sometimes fluctuated depending on hardware load and model behavior, and occasional hallucinations or incomplete outputs occurred.

## Limitations

CrewAI requires Python development skills to configure agents and define workflows effectively, as no fully graphical interface exists. Performance and resource requirements vary significantly depending on the chosen LLM and hardware.

Running multiple agents in parallel can be resource-intensive, sometimes causing stalled processes, incomplete results, or hallucinated outputs, particularly near token limits. The model may also ignore tool outputs or generate documentation for files that do not exist. Careful design of agent roles, task scope, prompt clarity, and tool usage is essential to mitigate these issues.