Get Started

Introduction

Build AI agent teams that work together to tackle complex tasks

What is CrewAI?

CrewAI is a cutting-edge framework for orchestrating autonomous AI agents.

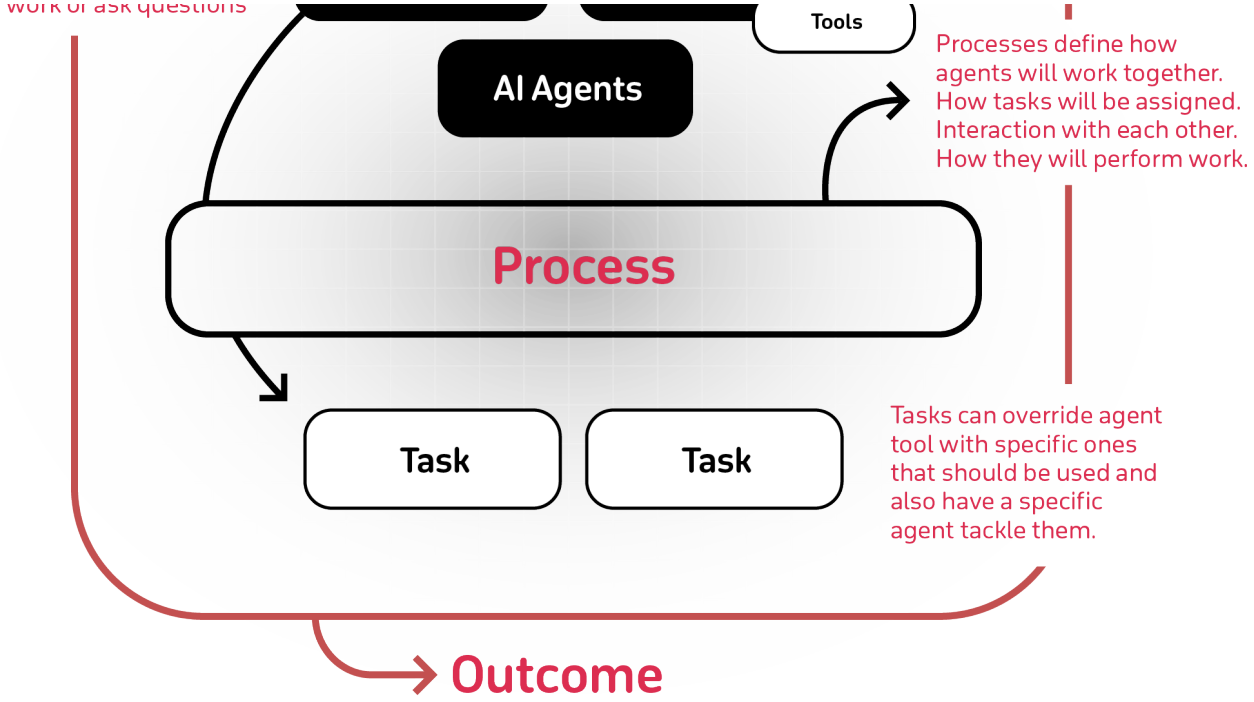
CrewAI enables you to create AI teams where each agent has specific roles, tools, and goals, working together to accomplish complex tasks.

Think of it as assembling your dream team - each member (agent) brings unique skills and expertise, collaborating seamlessly to achieve your objectives.

How CrewAI Works

Just like a company has departments (Sales, Engineering, Marketing) working together under leadership to achieve business goals, CrewAI helps you create an organization of AI agents with specialized roles collaborating to accomplish complex tasks.

Get Started › Introduction



CrewAI Framework Overview

Component	Description	Key Features
Crew	The top-level organization	• Manages AI agent teams • Oversees workflows • Ensures collaboration • Delivers outcomes
AI Agents	Specialized team members	• Have specific roles (researcher, writer) • Use designated tools • Can delegate tasks • Make autonomous decisions
Process	Workflow management system	• Defines collaboration patterns • Controls task assignments • Manages interactions • Ensures efficient execution
Tasks	Individual assignments	• Have clear objectives • Use specific tools • Feed into larger process • Produce actionable results

3. The Process ensures smooth collaboration
4. Tasks get completed to achieve the goal

Key Features

Role-Based Agents

Create specialized agents with defined roles, expertise, and goals - from researchers to analysts to writers

Flexible Tools

Equip agents with custom tools and APIs to interact with external services and data sources

Intelligent Collaboration

Agents work together, sharing insights and coordinating tasks to achieve complex objectives

Task Management

Define sequential or parallel workflows, with agents automatically handling task dependencies

Why Choose CrewAI?

- 🧠 **Autonomous Operation**: Agents make intelligent decisions based on their roles and available tools
- 📝 **Natural Interaction**: Agents communicate and collaborate like human team members
- 🛠️ **Extensible Design**: Easy to add new tools, roles, and capabilities
- 🚀 **Production Ready**: Built for reliability and scalability in real-world applications

your development environment.

guide to create your first CrewAI agent and get hands-on experience.

developers, get help, and share your CrewAI experiences.

Was this page helpful? Yes No

Installation ›

crewai

Get Started › Installation

Get Started

Installation

Get started with CrewAI - Install, configure, and build your first AI crew

Python Version Requirements

CrewAI requires `Python >=3.10 and <3.13` . Here's how to check your version:

```
python3 --version
```

If you need to update Python, visit python.org/downloads

Setting Up Your Environment

Before installing CrewAI, it's recommended to set up a virtual environment. This helps isolate your project dependencies and avoid conflicts.

1 **Create a Virtual Environment**

Choose your preferred method to create a virtual environment:

Using venv (Python's built-in tool):

Terminal

```
python3 -m venv .venv
```

Get Started › Installation

```
conda create -n crewai-env python=3.12
```

2 Activate the Virtual Environment

Activate your virtual environment based on your platform:

On macOS/Linux (venv):

Terminal

```
source .venv/bin/activate
```

On Windows (venv):

Terminal

```
.venv\Scripts\activate
```

Using conda (all platforms):

Terminal

```
conda activate crewai-env
```

Installing CrewAI

Now let's get you set up! 🚀

Get Started › Installation

Terminal

```
pip install 'crewai[tools]'
```

or

Terminal

```
pip install crewai crewai-tools
```

Both methods install the core package and additional tools needed for most use cases.

2 Upgrade CrewAI (Existing Installations Only)

If you have an older version of CrewAI installed, you can upgrade it:

Terminal

```
pip install --upgrade crewai crewai-tools
```

If you see a Poetry-related warning, you'll need to migrate to our new dependency manager:

Terminal

```
crewai update
```

3 Verify Installation

Check your installed versions:

Terminal

```
pip freeze | grep crewai
```

You should see something like:

Output

```
crewai==X.X.X
crewai-tools==X.X.X
```

Installation successful! You're ready to create your first crew.

Creating a New Project

We recommend using the YAML Template scaffolding for a structured approach to defining agents and tasks.

1 Generate Project Structure

Run the CrewAI CLI command:

Get Started › Installation

This creates a new project with the following structure:

```
my_project/
├── .gitignore
├── pyproject.toml
├── README.md
├── .env
└── src/
    └── my_project/
        ├── __init__.py
        ├── main.py
        ├── crew.py
        ├── tools/
        │   ├── custom_tool.py
        │   └── __init__.py
        └── config/
            ├── agents.yaml
            └── tasks.yaml
```

2 Install Additional Tools

You can install additional tools using UV:

Terminal

```
uv add <tool-name>
```

crewai

Get Started › Installation

Your project will contain these essential files:

File	Purpose
`agents.yaml`	Define your AI agents and their roles
`tasks.yaml`	Set up agent tasks and workflows
`.env`	Store API keys and environment variables
`main.py`	Project entry point and execution flow
`crew.py`	Crew orchestration and coordination
`tools/`	Directory for custom agent tools

Start by editing `agents.yaml` and `tasks.yaml` to define your crew's behavior. Keep sensitive information like API keys in `.env`.

Next Steps

Build Your First Agent

Follow our quickstart guide to create your first CrewAI agent and get hands-on experience.

Join the Community

Connect with other developers, get help, and share your CrewAI experiences.

Was this page helpful? Yes No

Powered by Mintlify

Get Started › Quickstart

Get Started

Quickstart

Build your first AI agent with CrewAI in under 5 minutes.

Build your first CrewAI Agent

Let's create a simple crew that will help us `research` and `report` on the `latest AI developments` for a given topic or subject.

Before we proceed, make sure you have `crewai` and `crewai-tools` installed. If you haven't installed them yet, you can do so by following the installation guide.

Follow the steps below to get crewing! 🚣

1 Create your crew

Create a new crew project by running the following command in your terminal. This will create a new directory called `latest-ai-development` with the basic structure for your crew.

Terminal

```
crewai create crew latest-ai-development
```

2 Modify your `agents.yaml` file

Get Started › Quickstart

agents.yaml

```yaml
# src/latest_ai_development/config/agents.yaml
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
    developments in {topic}. Known for your ability to find the most relev
    information and present it in a clear and concise manner.

reporting_analyst:
  role: >
    {topic} Reporting Analyst
  goal: >
    Create detailed reports based on {topic} data analysis and research fi
  backstory: >
    You're a meticulous analyst with a keen eye for detail. You're known f
    your ability to turn complex data into clear and concise reports, maki
    it easy for others to understand and act on the information you provid
```

3 Modify your `tasks.yaml` file

tasks.yaml

```yaml
# src/latest_ai_development/config/tasks.yaml
research_task:
  description: >
    Conduct a thorough research about {topic}
    Make sure you find any interesting and relevant information given
```

Get Started › Quickstart

```
reporting_task:
  description: >
    Review the context you got and expand each topic into a full section t
    Make sure the report is detailed and contains any and all relevant int
  expected_output: >
    A fully fledge reports with the mains topics, each with a full sectior
    Formatted as markdown without '```'
  agent: reporting_analyst
  output_file: report.md
```

4 Modify your `crew.py` file

crew.py

```python
# src/latest_ai_development/crew.py
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
from crewai_tools import SerperDevTool

@CrewBase
class LatestAiDevelopmentCrew():
  """LatestAiDevelopment crew"""

  @agent
  def researcher(self) -> Agent:
    return Agent(
      config=self.agents_config['researcher'],
      verbose=True,
      tools=[SerperDevTool()]
    )

  @agent
  def reporting_analyst(self) -> Agent:
    return Agent(
```

Get Started › Quickstart

```
  def research_task(self) -> Task:
    return Task(
      config=self.tasks_config['research_task'],
    )

  @task
  def reporting_task(self) -> Task:
    return Task(
      config=self.tasks_config['reporting_task'],
      output_file='output/report.md' # This is the file that will be conta
    )

  @crew
  def crew(self) -> Crew:
    """Creates the LatestAiDevelopment crew"""
    return Crew(
      agents=self.agents, # Automatically created by the @agent decorator
      tasks=self.tasks, # Automatically created by the @task decorator
      process=Process.sequential,
      verbose=True,
    )
```

5 [Optional] Add before and after crew functions

crew.py

```
# src/latest_ai_development/crew.py
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task, before_kickoff, af
from crewai_tools import SerperDevTool

@CrewBase
class LatestAiDevelopmentCrew():
  """LatestAiDevelopment crew"""
```

```
  @after_kickoff
  def after_kickoff_function(self, result):
    print(f"After kickoff function with result: {result}")
    return result # You can return the result or modify it as needed

  # ... remaining code
```

6 Feel free to pass custom inputs to your crew

For example, you can pass the `topic` input to your crew to customize the research and reporting.

main.py

```
#!/usr/bin/env python
# src/latest_ai_development/main.py
import sys
from latest_ai_development.crew import LatestAiDevelopmentCrew

def run():
  """
  Run the crew.
  """
  inputs = {
    'topic': 'AI Agents'
  }
  LatestAiDevelopmentCrew().crew().kickoff(inputs=inputs)
```

7 Set your environment variables

A Serper.dev API key: `SERPER_API_KEY=YOUR_KEY_HERE`

8 Lock and install the dependencies

Lock the dependencies and install them by using the CLI command but first, navigate to your project directory:

Terminal

```
cd latest-ai-development
crewai install
```

9 Run your crew

To run your crew, execute the following command in the root of your project:

Terminal

```
crewai run
```

10 View your final report

You should see the output in the console and the `report.md` file should be created in the root of your project with the final report.

Here's an example of what the report should look like:

output/report.md

crewai

Get Started › Quickstart

```
## 2. Benefits of AI Agents
AI agents bring numerous advantages that are transforming traditional wor

- **Task Automation**: AI agents can carry out repetitive tasks such as d
- **Improved Efficiency**: By quickly processing large datasets and perfo
- **Enhanced Decision-Making**: AI agents can analyze trends and patterns

## 3. Popular AI Agent Frameworks
Several frameworks have emerged to facilitate the development of AI agent

- **Autogen**: A framework designed to streamline the development of AI a
- **Semantic Kernel**: Focuses on natural language processing and underst
- **Promptflow**: Provides tools for developers to create conversational
- **Langchain**: Specializes in leveraging various APIs to ensure agents
- **CrewAI**: Aimed at collaborative environments, CrewAI strengthens tea
- **MemGPT**: Combines memory-optimized architectures with generative cap

These frameworks empower developers to build versatile and intelligent ag

## 4. AI Agents in Human Resources
AI agents are revolutionizing HR practices by automating and optimizing H

- **Recruiting**: AI agents can screen resumes, schedule interviews, and
- **Succession Planning**: AI systems analyze employee performance data a
- **Employee Engagement**: Chatbots powered by AI can facilitate feedback

As AI continues to evolve, HR departments leveraging these agents can rea

## 5. AI Agents in Finance
The finance sector is seeing extensive integration of AI agents that enha

- **Expense Tracking**: Automated systems manage and monitor expenses, fl
- **Risk Assessment**: AI models assess credit risk and uncover potential
- **Investment Decisions**: AI agents provide stock predictions and analy
```

Get Started › Quickstart

```
Conversely, corporations like Microsoft are taking strides to integrate A

## 7. Future Predictions and Implications
Experts predict that AI agents will transform essential aspects of work l

- Enhanced integration of AI agents across all business functions, creati
- Continued advancement of AI technologies, resulting in smarter, more ac
- Increased regulatory scrutiny to ensure ethical use, especially concern

To stay competitive and harness the full potential of AI agents, organiza

## 8. Conclusion
The emergence of AI agents is undeniably reshaping the workplace landscap
```

Note on Consistency in Naming

The names you use in your YAML files (`agents.yaml` and `tasks.yaml`) should match the method names in your Python code. For example, you can reference the agent for specific tasks from `tasks.yaml` file. This naming consistency allows CrewAI to automatically link your configurations with your code; otherwise, your task won't recognize the reference properly.

Example References

Note how we use the same name for the agent in the `agents.yaml` (`email_summarizer`) file as the method name in the `crew.py` (`email_summarizer`) file.

agents.yaml

```
    Summarize emails into a concise and clear summary
  backstory: >
    You will create a 5 bullet point summary of the report
  llm: openai/gpt-4o
```

Note how we use the same name for the agent in the `tasks.yaml` (`email_summarizer_task`) file as the method name in the `crew.py` (`email_summarizer_task`) file.

tasks.yaml

```
email_summarizer_task:
  description: >
    Summarize the email into a 5 bullet point summary
  expected_output: >
    A 5 bullet point summary of the email
  agent: email_summarizer
  context:
    - reporting_task
    - research_task
```

Use the annotations to properly reference the agent and task in the `crew.py` file.

Annotations include:

Here are examples of how to use each annotation in your CrewAI project, and when you should use them:

@agent

Used to define an agent in your crew. Use this when:

Get Started › Quickstart

```
@agent
def research_agent(self) -> Agent:
    return Agent(
        role="Research Analyst",
        goal="Conduct thorough research on given topics",
        backstory="Expert researcher with years of experience in data analysis",
        tools=[SerperDevTool()],
        verbose=True
    )
```

@task

Used to define a task that can be executed by agents. Use this when:

- You need to define a specific piece of work for an agent
- You want tasks to be automatically sequenced and managed
- You need to establish dependencies between different tasks

```
@task
def research_task(self) -> Task:
    return Task(
        description="Research the latest developments in AI technology",
        expected_output="A comprehensive report on AI advancements",
        agent=self.research_agent(),
        output_file="output/research.md"
    )
```

@crew

Used to define your crew configuration. Use this when:

```python
@crew
def research_crew(self) -> Crew:
    return Crew(
        agents=self.agents,  # Automatically collected from @agent methods
        tasks=self.tasks,    # Automatically collected from @task methods
        process=Process.sequential,
        verbose=True
    )
```

@tool

Used to create custom tools for your agents. Use this when:

- You need to give agents specific capabilities (like web search, data analysis)
- You want to encapsulate external API calls or complex operations
- You need to share functionality across multiple agents

```python
@tool
def web_search_tool(query: str, max_results: int = 5) -> list[str]:
    """
    Search the web for information.

    Args:
        query: The search query
        max_results: Maximum number of results to return

    Returns:
        List of search results
    """
    # Implement your search logic here
    return [f"Result {i} for: {query}" for i in range(max_results)]
```

You want to set up resources or configurations before execution

You need to perform any initialization logic

```
@before_kickoff
def validate_inputs(self, inputs: Optional[Dict[str, Any]]) -> Optional[Dict[str,
    """Validate and preprocess inputs before the crew starts."""
    if inputs is None:
        return None

    if 'topic' not in inputs:
        raise ValueError("Topic is required")

    # Add additional context
    inputs['timestamp'] = datetime.now().isoformat()
    inputs['topic'] = inputs['topic'].strip().lower()
    return inputs
```

@after_kickoff

Used to process results after the crew completes. Use this when:

You need to format or transform the final output

You want to perform cleanup operations

You need to save or log the results in a specific way

```
@after_kickoff
def process_results(self, result: CrewOutput) -> CrewOutput:
    """Process and format the results after the crew completes."""
    result.raw = result.raw.strip()
    result.raw = f"""
    # Research Results
```

@callback

Used to handle events during crew execution. Use this when:

- You need to monitor task progress
- You want to log intermediate results
- You need to implement custom progress tracking or metrics

```
@callback
def log_task_completion(self, task: Task, output: str):
    """Log task completion details for monitoring."""
    print(f"Task '{task.description}' completed")
    print(f"Output length: {len(output)} characters")
    print(f"Agent used: {task.agent.role}")
    print("-" * 50)
```

@cache_handler

Used to implement custom caching for task results. Use this when:

- You want to avoid redundant expensive operations
- You need to implement custom cache storage or expiration logic
- You want to persist results between runs

```
@cache_handler
def custom_cache(self, key: str) -> Optional[str]:
    """Custom cache implementation for storing task results."""
    cache_file = f"cache/{key}.json"
```

```
                return data['result']
    return None
```

These decorators are part of the CrewAI framework and help organize your crew's structure by automatically collecting agents, tasks, and handling various lifecycle events. They should be used within a class decorated with `@CrewBase`.

Replay Tasks from Latest Crew Kickoff

CrewAI now includes a replay feature that allows you to list the tasks from the last run and replay from a specific one. To use this feature, run.

```
crewai replay <task_id>
```

Replace `<task_id>` with the ID of the task you want to replay.

Reset Crew Memory

If you need to reset the memory of your crew before running it again, you can do so by calling the reset memory feature:

```
crewai reset-memories --all
```

This will clear the crew's memory, allowing for a fresh start.

Get Started › Quickstart

Deploy on Enterprise

Get started with CrewAI Enterprise and deploy your crew in a production environment with just a few clicks.

Join the Community

Join our open source community to discuss ideas, share your projects, and connect with other CrewAI developers.

Was this page helpful? Yes No

‹ Installation Agents ›

Core Concepts

Agents

Detailed guide on creating and managing agents within the CrewAI framework.

Overview of an Agent

In the CrewAI framework, an `Agent` is an autonomous unit that can:

- Perform specific tasks
- Make decisions based on its role and goal
- Use tools to accomplish objectives
- Communicate and collaborate with other agents
- Maintain memory of interactions
- Delegate tasks when allowed

> Think of an agent as a specialized team member with specific skills, expertise, and responsibilities. For example, a `Researcher` agent might excel at gathering and analyzing information, while a `Writer` agent might be better at creating content.

Agent Attributes

Attribute	Parameter	Type	Description
Role	`role`	`str`	Defines the agent's function and expertise within the crew.

Backstory	`backstory`	`str`	Provides context and personality to the agent, enriching interactions.
LLM *(optional)*	`llm`	`Union[str, LLM, Any]`	Language model that powers the agent. Defaults to the model specified in `OPENAI_MODEL_NAME` or "gpt-4".
Tools *(optional)*	`tools`	`List[BaseTool]`	Capabilities or functions available to the agent. Defaults to an empty list.
Function Calling LLM *(optional)*	`function_calling_llm`	`Optional[Any]`	Language model for tool calling, overrides crew's LLM if specified.
Max Iterations *(optional)*	`max_iter`	`int`	Maximum iterations before the agent must provide its best answer. Default is 20.
Max RPM *(optional)*	`max_rpm`	`Optional[int]`	Maximum requests per minute to avoid rate limits.
Max Execution Time *(optional)*	`max_execution_time`	`Optional[int]`	Maximum time (in seconds) for task execution.
Memory *(optional)*	`memory`	`bool`	Whether the agent should maintain memory of

(optional)			debugging. Default is False.
Allow Delegation *(optional)*	`allow_delegation`	`bool`	Allow the agent to delegate tasks to other agents. Default is False.
Step Callback *(optional)*	`step_callback`	`Optional[Any]`	Function called after each agent step, overrides crew callback.
Cache *(optional)*	`cache`	`bool`	Enable caching for tool usage. Default is True.
System Template *(optional)*	`system_template`	`Optional[str]`	Custom system prompt template for the agent.
Prompt Template *(optional)*	`prompt_template`	`Optional[str]`	Custom prompt template for the agent.
Response Template *(optional)*	`response_template`	`Optional[str]`	Custom response template for the agent.
Allow Code Execution *(optional)*	`allow_code_execution`	`Optional[bool]`	Enable code execution for the agent. Default is False.
Max Retry Limit *(optional)*	`max_retry_limit`	`int`	Maximum number of retries when an error occurs. Default is 2.

			Default is True.
Code Execution Mode *(optional)*	`code_execution_mode`	`Literal["safe", "unsafe"]`	Mode for code execution: 'safe' (using Docker) or 'unsafe' (direct). Default is 'safe'.
Embedder *(optional)*	`embedder`	`Optional[Dict[str, Any]]`	Configuration for the embedder used by the agent.
Knowledge Sources *(optional)*	`knowledge_sources`	`Optional[List[BaseKnowledgeSource]]`	Knowledge sources available to the agent.
Use System Prompt *(optional)*	`use_system_prompt`	`Optional[bool]`	Whether to use system prompt (for o1 model support). Default is True.

Creating Agents

There are two ways to create agents in CrewAI: using **YAML configuration (recommended)** or defining them **directly in code**.

YAML Configuration (Recommended)

Using YAML configuration provides a cleaner, more maintainable way to define agents. We strongly recommend using this approach in your CrewAI projects.

After creating your CrewAI project as outlined in the **Installation** section, navigate to the `src/latest_ai_development/config/agents.yaml` file and modify the template to match your requirements.

```
crew.kickoff(inputs={'topic': 'AI Agents'})
```

Here's an example of how to configure agents using YAML:

agents.yaml

```yaml
# src/latest_ai_development/config/agents.yaml
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
    developments in {topic}. Known for your ability to find the most relevant
    information and present it in a clear and concise manner.

reporting_analyst:
  role: >
    {topic} Reporting Analyst
  goal: >
    Create detailed reports based on {topic} data analysis and research findings
  backstory: >
    You're a meticulous analyst with a keen eye for detail. You're known for
    your ability to turn complex data into clear and concise reports, making
    it easy for others to understand and act on the information you provide.
```

To use this YAML configuration in your code, create a crew class that inherits from `CrewBase`:

```python
from crewai.project import CrewBase, agent, crew
from crewai_tools import SerperDevTool

@CrewBase
class LatestAiDevelopmentCrew():
  """LatestAiDevelopment crew"""

  agents_config = "config/agents.yaml"

  @agent
  def researcher(self) -> Agent:
    return Agent(
      config=self.agents_config['researcher'],
      verbose=True,
      tools=[SerperDevTool()]
    )

  @agent
  def reporting_analyst(self) -> Agent:
    return Agent(
      config=self.agents_config['reporting_analyst'],
      verbose=True
    )
```

The names you use in your YAML files (`agents.yaml`) should match the method names in your Python code.

Direct Code Definition

You can create agents directly in code by instantiating the `Agent` class. Here's a comprehensive example showing all available parameters:

```
# Create an agent with all available parameters
agent = Agent(
    role="Senior Data Scientist",
    goal="Analyze and interpret complex datasets to provide actionable insights",
    backstory="With over 10 years of experience in data science and machine learn
              "you excel at finding patterns in complex datasets.",
    llm="gpt-4",  # Default: OPENAI_MODEL_NAME or "gpt-4"
    function_calling_llm=None,  # Optional: Separate LLM for tool calling
    memory=True,  # Default: True
    verbose=False,  # Default: False
    allow_delegation=False,  # Default: False
    max_iter=20,  # Default: 20 iterations
    max_rpm=None,  # Optional: Rate limit for API calls
    max_execution_time=None,  # Optional: Maximum execution time in seconds
    max_retry_limit=2,  # Default: 2 retries on error
    allow_code_execution=False,  # Default: False
    code_execution_mode="safe",  # Default: "safe" (options: "safe", "unsafe")
    respect_context_window=True,  # Default: True
    use_system_prompt=True,  # Default: True
    tools=[SerperDevTool()],  # Optional: List of tools
    knowledge_sources=None,  # Optional: List of knowledge sources
    embedder=None,  # Optional: Custom embedder configuration
    system_template=None,  # Optional: Custom system prompt template
    prompt_template=None,  # Optional: Custom prompt template
    response_template=None,  # Optional: Custom response template
    step_callback=None,  # Optional: Callback function for monitoring
)
```

Let’s break down some key parameter combinations for common use cases:

Basic Research Agent

```
    goal="Find and summarize information about specific topics",
    backstory="You are an experienced researcher with attention to detail",
    tools=[SerperDevTool()],
    verbose=True  # Enable logging for debugging
)
```

Code Development Agent

Code

```
dev_agent = Agent(
    role="Senior Python Developer",
    goal="Write and debug Python code",
    backstory="Expert Python developer with 10 years of experience",
    allow_code_execution=True,
    code_execution_mode="safe",  # Uses Docker for safety
    max_execution_time=300,  # 5-minute timeout
    max_retry_limit=3  # More retries for complex code tasks
)
```

Long-Running Analysis Agent

Code

```
analysis_agent = Agent(
    role="Data Analyst",
    goal="Perform deep analysis of large datasets",
    backstory="Specialized in big data analysis and pattern recognition",
    memory=True,
    respect_context_window=True,
    max_rpm=10,  # Limit API calls
```

Custom Template Agent

Code

```
custom_agent = Agent(
    role="Customer Service Representative",
    goal="Assist customers with their inquiries",
    backstory="Experienced in customer support with a focus on satisfaction",
    system_template="""<|start_header_id|>system<|end_header_id|>
                       {{ .System }}<|eot_id|>""",
    prompt_template="""<|start_header_id|>user<|end_header_id|>
                       {{ .Prompt }}<|eot_id|>""",
    response_template="""<|start_header_id|>assistant<|end_header_id|>
                       {{ .Response }}<|eot_id|>""",
)
```

Parameter Details

Critical Parameters

`role` , `goal` , and `backstory` are required and shape the agent's behavior

`llm` determines the language model used (default: OpenAI's GPT-4)

Memory and Context

`memory` : Enable to maintain conversation history

`respect_context_window` : Prevents token limit issues

`knowledge_sources` : Add domain-specific knowledge bases

Execution Control

`max_iter` : Maximum attempts before giving best answer

Code Execution

`allow_code_execution` : Must be True to run code

`code_execution_mode` :

`"safe"` : Uses Docker (recommended for production)

`"unsafe"` : Direct execution (use only in trusted environments)

Templates

`system_template` : Defines agent's core behavior

`prompt_template` : Structures input format

`response_template` : Formats agent responses

When using custom templates, you can use variables like `{role}` , `{goal}` , and `{input}` in your templates. These will be automatically populated during execution.

Agent Tools

Agents can be equipped with various tools to enhance their capabilities. CrewAI supports tools from:

CrewAI Toolkit

LangChain Tools

Here's how to add tools to an agent:

Code

```
search_tool = SerperDevTool()
wiki_tool = WikipediaTools()

# Add tools to agent
researcher = Agent(
    role="AI Technology Researcher",
    goal="Research the latest AI developments",
    tools=[search_tool, wiki_tool],
    verbose=True
)
```

Agent Memory and Context

Agents can maintain memory of their interactions and use context from previous tasks. This is particularly useful for complex workflows where information needs to be retained across multiple tasks.

Code

```
from crewai import Agent

analyst = Agent(
    role="Data Analyst",
    goal="Analyze and remember complex data patterns",
    memory=True,  # Enable memory
    verbose=True
)
```

When `memory` is enabled, the agent will maintain context across multiple interactions, improving its ability to handle complex, multi-step tasks.

- Use `code_execution_mode: "safe"` (Docker) in production environments
- Consider setting appropriate `max_execution_time` limits to prevent infinite loops

Performance Optimization

- Use `respect_context_window: true` to prevent token limit issues
- Set appropriate `max_rpm` to avoid rate limiting
- Enable `cache: true` to improve performance for repetitive tasks
- Adjust `max_iter` and `max_retry_limit` based on task complexity

Memory and Context Management

- Use `memory: true` for tasks requiring historical context
- Leverage `knowledge_sources` for domain-specific information
- Configure `embedder_config` when using custom embedding models
- Use custom templates (`system_template` , `prompt_template` , `response_template`) for fine-grained control over agent behavior

Agent Collaboration

- Enable `allow_delegation: true` when agents need to work together
- Use `step_callback` to monitor and log agent interactions
- Consider using different LLMs for different purposes:
  - Main `llm` for complex reasoning
  - `function_calling_llm` for efficient tool usage

Troubleshooting Common Issues

1. **Rate Limiting**: If you're hitting API rate limits:
   - Implement appropriate `max_rpm`
   - Use caching for repetitive operations
   - Consider batching requests
2. **Context Window Errors**: If you're exceeding context limits:
   - Enable `respect_context_window`
   - Use more efficient prompts
   - Clear agent memory periodically
3. **Code Execution Issues**: If code execution fails:
   - Verify Docker is installed for safe mode
   - Check execution permissions
   - Review code sandbox settings
4. **Memory Issues**: If agent responses seem inconsistent:
   - Verify memory is enabled
   - Check knowledge source configuration
   - Review conversation history management

Remember that agents are most effective when configured according to their specific use case. Take time to understand your requirements and adjust these parameters accordingly.

Core Concepts

Tasks

Detailed guide on managing and creating tasks within the CrewAI framework.

Overview of a Task

In the CrewAI framework, a `Task` is a specific assignment completed by an `Agent`.

Tasks provide all necessary details for execution, such as a description, the agent responsible, required tools, and more, facilitating a wide range of action complexities.

Tasks within CrewAI can be collaborative, requiring multiple agents to work together. This is managed through the task properties and orchestrated by the Crew's process, enhancing teamwork and efficiency.

Task Execution Flow

Tasks can be executed in two ways:

- **Sequential**: Tasks are executed in the order they are defined
- **Hierarchical**: Tasks are assigned to agents based on their roles and expertise

The execution flow is defined when creating the crew:

Code

```
crew = Crew(
    agents=[agent1, agent2],
    tasks=[task1, task2],
```

Core Concepts › Tasks

Task Attributes

Attribute	Parameters	Type	Description
Description	`description`	`str`	A clear, concise statement of what the task entails.
Expected Output	`expected_output`	`str`	A detailed description of what the task's completion looks like.
Name *(optional)*	`name`	`Optional[str]`	A name identifier for the task.
Agent *(optional)*	`agent`	`Optional[BaseAgent]`	The agent responsible for executing the task.
Tools *(optional)*	`tools`	`List[BaseTool]`	The tools/resources the agent is limited to use for this task.
Context *(optional)*	`context`	`Optional[List["Task"]]`	Other tasks whose outputs will be used as context for this task.
Async Execution *(optional)*	`async_execution`	`Optional[bool]`	Whether the task should be executed asynchronously. Defaults to False.
Human Input *(optional)*	`human_input`	`Optional[bool]`	Whether the task should have a human review the final answer of the agent. Defaults to False.
Config *(optional)*	`config`	`Optional[Dict[str, Any]]`	Task-specific configuration parameters.
Output File *(optional)*	`output_file`	`Optional[str]`	File path for storing the task output.
Output JSON *(optional)*	`output_json`	`Optional[Type[BaseModel]]`	A Pydantic model to structure the JSON output.
Output Pydantic *(optional)*	`output_pydantic`	`Optional[Type[BaseModel]]`	A Pydantic model for task output.

Core Concepts › Tasks

Creating Tasks

There are two ways to create tasks in CrewAI: using **YAML configuration (recommended)** or defining them **directly in code**.

YAML Configuration (Recommended)

Using YAML configuration provides a cleaner, more maintainable way to define tasks. We strongly recommend using this approach to define tasks in your CrewAI projects.

After creating your CrewAI project as outlined in the **Installation** section, navigate to the `src/latest_ai_development/config/tasks.yaml` file and modify the template to match your specific task requirements.

> Variables in your YAML files (like `{topic}`) will be replaced with values from your inputs when running the crew:
>
> Code
>
> ```
> crew.kickoff(inputs={'topic': 'AI Agents'})
> ```

Here's an example of how to configure tasks using YAML:

tasks.yaml

```
research_task:
  description: >
    Conduct a thorough research about {topic}
    Make sure you find any interesting and relevant information given
```

```
reporting_task:
  description: >
    Review the context you got and expand each topic into a full section for a re
    Make sure the report is detailed and contains any and all relevant informatio
  expected_output: >
    A fully fledge reports with the mains topics, each with a full section of inf
    Formatted as markdown without '```'
  agent: reporting_analyst
  output_file: report.md
```

To use this YAML configuration in your code, create a crew class that inherits from `CrewBase`:

crew.py

```python
# src/latest_ai_development/crew.py

from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
from crewai_tools import SerperDevTool

@CrewBase
class LatestAiDevelopmentCrew():
  """LatestAiDevelopment crew"""

  @agent
  def researcher(self) -> Agent:
    return Agent(
      config=self.agents_config['researcher'],
      verbose=True,
      tools=[SerperDevTool()]
    )
```

```
    )

  @task
  def research_task(self) -> Task:
    return Task(
      config=self.tasks_config['research_task']
    )

  @task
  def reporting_task(self) -> Task:
    return Task(
      config=self.tasks_config['reporting_task']
    )

  @crew
  def crew(self) -> Crew:
    return Crew(
      agents=[
        self.researcher(),
        self.reporting_analyst()
      ],
      tasks=[
        self.research_task(),
        self.reporting_task()
      ],
      process=Process.sequential
    )
```

The names you use in your YAML files (`agents.yaml` and `tasks.yaml`) should match the method names in your Python code.

Direct Code Definition (Alternative)

```
from crewai import Task

research_task = Task(
    description="""
        Conduct a thorough research about AI Agents.
        Make sure you find any interesting and relevant information given
        the current year is 2025.
    """,
    expected_output="""
        A list with 10 bullet points of the most relevant information about AI Ag
    """,
    agent=researcher
)

reporting_task = Task(
    description="""
        Review the context you got and expand each topic into a full section for
        Make sure the report is detailed and contains any and all relevant inform
    """,
    expected_output="""
        A fully fledge reports with the mains topics, each with a full section of
        Formatted as markdown without '```'
    """,
    agent=reporting_analyst,
    output_file="report.md"
)
```

Directly specify an `agent` for assignment or let the `hierarchical` CrewAI's process decide based on roles, availability, etc.

Task Output

The output of a task in CrewAI framework is encapsulated within the `TaskOutput` class. This class provides a structured way to access results of a task, including various formats such as raw output, JSON, and Pydantic models.

By default, the `TaskOutput` will only include the `raw` output. A `TaskOutput` will only include the `pydantic` or `json_dict` output if the original `Task` object was configured with `output_pydantic` or `output_json`, respectively.

Task Output Attributes

Attribute	Parameters	Type	Description
Description	`description`	`str`	Description of the task.
Summary	`summary`	`Optional[str]`	Summary of the task, auto-generated from the first 10 words of the description.
Raw	`raw`	`str`	The raw output of the task. This is the default format for the output.
Pydantic	`pydantic`	`Optional[BaseModel]`	A Pydantic model object representing the structured output of the task.
JSON Dict	`json_dict`	`Optional[Dict[str, Any]]`	A dictionary representing the JSON output of the task.
Agent	`agent`	`str`	The agent that executed the task.
Output Format	`output_format`	`OutputFormat`	The format of the task output, with options including RAW, JSON, and Pydantic. The default is RAW.

Task Methods and Properties

Method/Property	Description
json	Returns the JSON string representation of the task output if the output format is JSON.

Accessing Task Outputs

Once a task has been executed, its output can be accessed through the `output` attribute of the `Task` object. The `TaskOutput` class provides various ways to interact with and present this output.

Example

Code

```
# Example task
task = Task(
    description='Find and summarize the latest AI news',
    expected_output='A bullet list summary of the top 5 most important AI news',
    agent=research_agent,
    tools=[search_tool]
)

# Execute the crew
crew = Crew(
    agents=[research_agent],
    tasks=[task],
    verbose=True
)

result = crew.kickoff()

# Accessing the task output
task_output = task.output

print(f"Task Description: {task_output.description}")
print(f"Task Summary: {task_output.summary}")
```

Task Dependencies and Context

Tasks can depend on the output of other tasks using the `context` attribute. For example:

Code

```
research_task = Task(
    description="Research the latest developments in AI",
    expected_output="A list of recent AI developments",
    agent=researcher
)

analysis_task = Task(
    description="Analyze the research findings and identify key trends",
    expected_output="Analysis report of AI trends",
    agent=analyst,
    context=[research_task]  # This task will wait for research_task to complete
)
```

Task Guardrails

Task guardrails provide a way to validate and transform task outputs before they are passed to the next task. This feature helps ensure data quality and provides feedback to agents when their output doesn't meet specific criteria.

Using Task Guardrails

```python
from typing import Tuple, Union, Dict, Any

def validate_blog_content(result: str) -> Tuple[bool, Union[Dict[str, Any], str]]:
    """Validate blog content meets requirements."""
    try:
        # Check word count
        word_count = len(result.split())
        if word_count > 200:
            return (False, {
                "error": "Blog content exceeds 200 words",
                "code": "WORD_COUNT_ERROR",
                "context": {"word_count": word_count}
            })

        # Additional validation logic here
        return (True, result.strip())
    except Exception as e:
        return (False, {
            "error": "Unexpected error during validation",
            "code": "SYSTEM_ERROR"
        })

blog_task = Task(
    description="Write a blog post about AI",
    expected_output="A blog post under 200 words",
    agent=blog_agent,
    guardrail=validate_blog_content  # Add the guardrail function
)
```

Guardrail Function Requirements

1. Function Signature:

2. **Return Values**:

   Success: Return `(True, validated_result)`

   Failure: Return `(False, error_details)`

Error Handling Best Practices

1. **Structured Error Responses**:

Code

```
def validate_with_context(result: str) -> Tuple[bool, Union[Dict[str, Any], str]]
    try:
        # Main validation logic
        validated_data = perform_validation(result)
        return (True, validated_data)
    except ValidationError as e:
        return (False, {
            "error": str(e),
            "code": "VALIDATION_ERROR",
            "context": {"input": result}
        })
    except Exception as e:
        return (False, {
            "error": "Unexpected error",
            "code": "SYSTEM_ERROR"
        })
```

2. **Error Categories**:

   Use specific error codes

Code

```python
from typing import Any, Dict, List, Tuple, Union

def complex_validation(result: str) -> Tuple[bool, Union[str, Dict[str, Any]]]:
    """Chain multiple validation steps."""
    # Step 1: Basic validation
    if not result:
        return (False, {"error": "Empty result", "code": "EMPTY_INPUT"})

    # Step 2: Content validation
    try:
        validated = validate_content(result)
        if not validated:
            return (False, {"error": "Invalid content", "code": "CONTENT_ERROR"})

        # Step 3: Format validation
        formatted = format_output(validated)
        return (True, formatted)
    except Exception as e:
        return (False, {
            "error": str(e),
            "code": "VALIDATION_ERROR",
            "context": {"step": "content_validation"}
        })
```

Handling Guardrail Results

When a guardrail returns `(False, error)`:

1. The error is sent back to the agent
2. The agent attempts to fix the issue

Example with retry handling:

Code

```
from typing import Optional, Tuple, Union

def validate_json_output(result: str) -> Tuple[bool, Union[Dict[str, Any], str]]:
    """Validate and parse JSON output."""
    try:
        # Try to parse as JSON
        data = json.loads(result)
        return (True, data)
    except json.JSONDecodeError as e:
        return (False, {
            "error": "Invalid JSON format",
            "code": "JSON_ERROR",
            "context": {"line": e.lineno, "column": e.colno}
        })

task = Task(
    description="Generate a JSON report",
    expected_output="A valid JSON object",
    agent=analyst,
    guardrail=validate_json_output,
    max_retries=3  # Limit retry attempts
)
```

Getting Structured Consistent Outputs from Tasks

It's also important to note that the output of the final task of a crew becomes the final output of the actual crew itself.

Here's an example demonstrating how to use output_pydantic:

Code

```
import json

from crewai import Agent, Crew, Process, Task
from pydantic import BaseModel


class Blog(BaseModel):
    title: str
    content: str


blog_agent = Agent(
    role="Blog Content Generator Agent",
    goal="Generate a blog title and content",
    backstory="""You are an expert content creator, skilled in crafting engaging
    verbose=False,
    allow_delegation=False,
    llm="gpt-4o",
)

task1 = Task(
    description="""Create a blog title and content on a given topic. Make sure th
    expected_output="A compelling blog title and well-written content.",
    agent=blog_agent,
    output_pydantic=Blog,
)

# Instantiate your crew with a sequential process
crew = Crew(
    agents=[blog_agent],
```

```
result = crew.kickoff()

# Option 1: Accessing Properties Using Dictionary-Style Indexing
print("Accessing Properties - Option 1")
title = result["title"]
content = result["content"]
print("Title:", title)
print("Content:", content)

# Option 2: Accessing Properties Directly from the Pydantic Model
print("Accessing Properties - Option 2")
title = result.pydantic.title
content = result.pydantic.content
print("Title:", title)
print("Content:", content)

# Option 3: Accessing Properties Using the to_dict() Method
print("Accessing Properties - Option 3")
output_dict = result.to_dict()
title = output_dict["title"]
content = output_dict["content"]
print("Title:", title)
print("Content:", content)

# Option 4: Printing the Entire Blog Object
print("Accessing Properties - Option 5")
print("Blog:", result)
```

In this example:

A Pydantic model Blog is defined with title and content fields.

Explanation of Accessing the Output

1. Dictionary-Style Indexing: You can directly access the fields using result["field_name"]. This works because the CrewOutput class implements the getitem method.
2. Directly from Pydantic Model: Access the attributes directly from the result.pydantic object.
3. Using to_dict() Method: Convert the output to a dictionary and access the fields.
4. Printing the Entire Object: Simply print the result object to see the structured output.

Using `output_json`

The `output_json` property allows you to define the expected output in JSON format. This ensures that the task's output is a valid JSON structure that can be easily parsed and used in your application.

Here's an example demonstrating how to use `output_json` :

Code

```
import json

from crewai import Agent, Crew, Process, Task
from pydantic import BaseModel


# Define the Pydantic model for the blog
class Blog(BaseModel):
    title: str
    content: str
```

```
    verbose=False,
    allow_delegation=False,
    llm="gpt-4o",
)

# Define the task with output_json set to the Blog model
task1 = Task(
    description="""Create a blog title and content on a given topic. Make sure th
    expected_output="A JSON object with 'title' and 'content' fields.",
    agent=blog_agent,
    output_json=Blog,
)

# Instantiate the crew with a sequential process
crew = Crew(
    agents=[blog_agent],
    tasks=[task1],
    verbose=True,
    process=Process.sequential,
)

# Kickoff the crew to execute the task
result = crew.kickoff()

# Option 1: Accessing Properties Using Dictionary-Style Indexing
print("Accessing Properties - Option 1")
title = result["title"]
content = result["content"]
print("Title:", title)
print("Content:", content)

# Option 2: Printing the Entire Blog Object
print("Accessing Properties - Option 2")
print("Blog:", result)
```

The task task1 uses the output_json property to indicate that it expects a JSON output conforming to the Blog model.

After executing the crew, you can access the structured JSON output in two ways as shown.

Explanation of Accessing the Output

1. Accessing Properties Using Dictionary-Style Indexing: You can access the fields directly using result["field_name"]. This is possible because the CrewOutput class implements the getitem method, allowing you to treat the output like a dictionary. In this option, we're retrieving the title and content from the result.
2. Printing the Entire Blog Object: By printing result, you get the string representation of the CrewOutput object. Since the str method is implemented to return the JSON output, this will display the entire output as a formatted string representing the Blog object.

By using output_pydantic or output_json, you ensure that your tasks produce outputs in a consistent and structured format, making it easier to process and utilize the data within your application or across multiple tasks.

Integrating Tools with Tasks

Leverage tools from the CrewAI Toolkit and LangChain Tools for enhanced task performance and agent interaction.

Creating a Task with Tools

```
os.environ["SERPER_API_KEY"] = "Your Key" # serper.dev API key

from crewai import Agent, Task, Crew
from crewai_tools import SerperDevTool

research_agent = Agent(
  role='Researcher',
  goal='Find and summarize the latest AI news',
  backstory="""You're a researcher at a large company.
  You're responsible for analyzing data and providing insights
  to the business.""",
  verbose=True
)

# to perform a semantic search for a specified query from a text's content across
search_tool = SerperDevTool()

task = Task(
  description='Find and summarize the latest AI news',
  expected_output='A bullet list summary of the top 5 most important AI news',
  agent=research_agent,
  tools=[search_tool]
)

crew = Crew(
    agents=[research_agent],
    tasks=[task],
    verbose=True
)

result = crew.kickoff()
print(result)
```

Referring to Other Tasks

In CrewAI, the output of one task is automatically relayed into the next one, but you can specifically define what tasks' output, including multiple, should be used as context for another task.

This is useful when you have a task that depends on the output of another task that is not performed immediately after it. This is done through the `context` attribute of the task:

Code

```
# ...

research_ai_task = Task(
    description="Research the latest developments in AI",
    expected_output="A list of recent AI developments",
    async_execution=True,
    agent=research_agent,
    tools=[search_tool]
)

research_ops_task = Task(
    description="Research the latest developments in AI Ops",
    expected_output="A list of recent AI Ops developments",
    async_execution=True,
    agent=research_agent,
    tools=[search_tool]
)

write_blog_task = Task(
    description="Write a full blog post about the importance of AI and its latest
    expected_output="Full blog post that is 4 paragraphs long",
    agent=writer_agent,
    context=[research_ai_task, research_ops_task]
)
```

Asynchronous Execution

You can define a task to be executed asynchronously. This means that the crew will not wait for it to be completed to continue with the next task. This is useful for tasks that take a long time to be completed, or that are not crucial for the next tasks to be performed.

You can then use the `context` attribute to define in a future task that it should wait for the output of the asynchronous task to be completed.

Code

```
#...

list_ideas = Task(
    description="List of 5 interesting ideas to explore for an article about AI."
    expected_output="Bullet point list of 5 ideas for an article.",
    agent=researcher,
    async_execution=True # Will be executed asynchronously
)

list_important_history = Task(
    description="Research the history of AI and give me the 5 most important even
    expected_output="Bullet point list of 5 important events.",
    agent=researcher,
    async_execution=True # Will be executed asynchronously
)

write_article = Task(
    description="Write an article about AI, its history, and interesting ideas.",
    expected_output="A 4 paragraph article about AI.",
    agent=writer,
    context=[list_ideas, list_important_history] # Will wait for the output of th
)
```

Callback Mechanism

The callback function is executed after the task is completed, allowing for actions or notifications to be triggered based on the task's outcome.

Code

```
# ...

def callback_function(output: TaskOutput):
    # Do something after the task is completed
    # Example: Send an email to the manager
    print(f"""
        Task completed!
        Task: {output.description}
        Output: {output.raw}
    """)

research_task = Task(
    description='Find and summarize the latest AI news',
    expected_output='A bullet list summary of the top 5 most important AI news',
    agent=research_agent,
    tools=[search_tool],
    callback=callback_function
)

#...
```

Accessing a Specific Task Output

Once a crew finishes running, you can access the output of a specific task by using the `output` attribute of the task object:

```
    description='Find and summarize the latest AI news',
    expected_output='A bullet list summary of the top 5 most important AI news',
    agent=research_agent,
    tools=[search_tool]
)

#...

crew = Crew(
    agents=[research_agent],
    tasks=[task1, task2, task3],
    verbose=True
)

result = crew.kickoff()

# Returns a TaskOutput object with the description and results of the task
print(f"""
    Task completed!
    Task: {task1.output.description}
    Output: {task1.output.raw}
""")
```

Tool Override Mechanism

Specifying tools in a task allows for dynamic adaptation of agent capabilities, emphasizing CrewAI's flexibility.

Error Handling and Validation Mechanisms

While creating and executing tasks, certain validation mechanisms are in place to ensure the robustness and reliability of task attributes. These include but are not limited to:

These validations help in maintaining the consistency and reliability of task executions within the crewAI framework.

Task Guardrails

Task guardrails provide a powerful way to validate, transform, or filter task outputs before they are passed to the next task. Guardrails are optional functions that execute before the next task starts, allowing you to ensure that task outputs meet specific requirements or formats.

Basic Usage

Code

```
from typing import Tuple, Union
from crewai import Task

def validate_json_output(result: str) -> Tuple[bool, Union[dict, str]]:
    """Validate that the output is valid JSON."""
    try:
        json_data = json.loads(result)
        return (True, json_data)
    except json.JSONDecodeError:
        return (False, "Output must be valid JSON")

task = Task(
    description="Generate JSON data",
    expected_output="Valid JSON object",
    guardrail=validate_json_output
)
```

2. Execution Timing: The guardrail function is executed before the next task starts, ensuring valid data flow between tasks.
3. Return Format: Guardrails must return a tuple of `(success, data)`:
   - If `success` is `True`, `data` is the validated/transformed result
   - If `success` is `False`, `data` is the error message
4. Result Routing:
   - On success (`True`), the result is automatically passed to the next task
   - On failure (`False`), the error is sent back to the agent to generate a new answer

Common Use Cases

Data Format Validation

Code

```
def validate_email_format(result: str) -> Tuple[bool, Union[str, str]]:
    """Ensure the output contains a valid email address."""
    import re
    email_pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    if re.match(email_pattern, result.strip()):
        return (True, result.strip())
    return (False, "Output must be a valid email address")
```

Content Filtering

Code

```
def filter_sensitive_info(result: str) -> Tuple[bool, Union[str, str]]:
    """Remove or validate sensitive information."""
```

Data Transformation

Code

```
def normalize_phone_number(result: str) -> Tuple[bool, Union[str, str]]:
    """Ensure phone numbers are in a consistent format."""
    import re
    digits = re.sub(r'\D', '', result)
    if len(digits) == 10:
        formatted = f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
        return (True, formatted)
    return (False, "Output must be a 10-digit phone number")
```

Advanced Features

Chaining Multiple Validations

Code

```
def chain_validations(*validators):
    """Chain multiple validators together."""
    def combined_validator(result):
        for validator in validators:
            success, data = validator(result)
            if not success:
                return (False, data)
            result = data
        return (True, result)
    return combined_validator
```

```
        validate_email_format,
        filter_sensitive_info
    )
)
```

Custom Retry Logic

Code

```
task = Task(
    description="Generate data",
    expected_output="Valid data",
    guardrail=validate_data,
    max_retries=5  # Override default retry limit
)
```

Creating Directories when Saving Files

You can now specify if a task should create directories when saving its output to a file. This is particularly useful for organizing outputs and ensuring that file paths are correctly structured.

Code

```
# ...

save_output_task = Task(
    description='Save the summarized AI news to a file',
    expected_output='File saved successfully',
    agent=research_agent,
    tools=[file_save_tool],
```

Conclusion

Tasks are the driving force behind the actions of agents in CrewAI. By properly defining tasks and their outcomes, you set the stage for your AI agents to work effectively, either independently or as a collaborative unit. Equipping tasks with appropriate tools, understanding the execution process, and following robust validation practices are crucial for maximizing CrewAI's potential, ensuring agents are effectively prepared for their assignments and that tasks are executed as intended.

‹ Agents Crews ›

Core Concepts

Crews

Understanding and utilizing crews in the crewAI framework with comprehensive attributes and functionalities.

What is a Crew?

A crew in crewAI represents a collaborative group of agents working together to achieve a set of tasks. Each crew defines the strategy for task execution, agent collaboration, and the overall workflow.

Crew Attributes

Attribute	Parameters	Description
Tasks	`tasks`	A list of tasks assigned to the crew.
Agents	`agents`	A list of agents that are part of the crew.
Process *(optional)*	`process`	The process flow (e.g., sequential, hierarchical) the crew follows. Default is `sequential`.
Verbose *(optional)*	`verbose`	The verbosity level for logging during execution. Defaults to `False`.
Manager LLM *(optional)*	`manager_llm`	The language model used by the manager agent in a hierarchical process. **Required when using a hierarchical process.**
Function Calling LLM *(optional)*	`function_calling_llm`	If passed, the crew will use this LLM to do function calling for tools for all agents in the crew. Each agent can have its own LLM, which overrides the crew's LLM for function calling.

(optional)		execution. Defaults to `None`.
Language *(optional)*	`language`	Language used for the crew, defaults to English.
Language File *(optional)*	`language_file`	Path to the language file to be used for the crew.
Memory *(optional)*	`memory`	Utilized for storing execution memories (short-term, long-term, entity memory).
Memory Config *(optional)*	`memory_config`	Configuration for the memory provider to be used by the crew.
Cache *(optional)*	`cache`	Specifies whether to use a cache for storing the results of tools' execution. Defaults to `True`.
Embedder *(optional)*	`embedder`	Configuration for the embedder to be used by the crew. Mostly used by memory for now. Default is `{"provider": "openai"}`.
Full Output *(optional)*	`full_output`	Whether the crew should return the full output with all tasks outputs or just the final output. Defaults to `False`.
Step Callback *(optional)*	`step_callback`	A function that is called after each step of every agent. This can be used to log the agent's actions or to perform other operations; it won't override the agent-specific `step_callback`.
Task Callback *(optional)*	`task_callback`	A function that is called after the completion of each task. Useful for monitoring or additional operations post-task execution.
Share Crew *(optional)*	`share_crew`	Whether you want to share the complete crew information and execution with the crewAI team to make the library better, and allow us to train models.
Output Log File *(optional)*	`output_log_file`	Set to True to save logs as logs.txt in the current directory or provide a file path. Logs will be in JSON format if the filename ends in .json, otherwise .txt. Defautls to `None`.
Manager Agent *(optional)*	`manager_agent`	`manager` sets a custom agent that will be used as a manager.

(optional)		each Crew iteration, all Crew data is sent to an AgentPlanner that will plan the tasks and this plan will be added to each task description.
Planning LLM *(optional)*	`planning_llm`	The language model used by the AgentPlanner in a planning process.

> **Crew Max RPM:** The `max_rpm` attribute sets the maximum number of requests per minute the crew can perform to avoid rate limits and will override individual agents' `max_rpm` settings if you set it.

Creating Crews

There are two ways to create crews in CrewAI: using **YAML configuration (recommended)** or defining them **directly in code**.

YAML Configuration (Recommended)

Using YAML configuration provides a cleaner, more maintainable way to define crews and is consistent with how agents and tasks are defined in CrewAI projects.

After creating your CrewAI project as outlined in the **Installation** section, you can define your crew in a class that inherits from `CrewBase` and uses decorators to define agents, tasks, and the crew itself.

Example Crew Class with Decorators

code

```
from crewai import Agent, Crew, Task, Process
from crewai.project import CrewBase, agent, task, crew, before_kickoff, after_kic
```

```
    # Paths to your YAML configuration files
    # To see an example agent and task defined in YAML, checkout the following:
    # - Task: https://docs.crewai.com/concepts/tasks#yaml-configuration-recommend
    # - Agents: https://docs.crewai.com/concepts/agents#yaml-configuration-recomm
    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @before_kickoff
    def prepare_inputs(self, inputs):
        # Modify inputs before the crew starts
        inputs['additional_data'] = "Some extra information"
        return inputs

    @after_kickoff
    def process_output(self, output):
        # Modify output after the crew finishes
        output.raw += "\nProcessed after kickoff."
        return output

    @agent
    def agent_one(self) -> Agent:
        return Agent(
            config=self.agents_config['agent_one'],
            verbose=True
        )

    @agent
    def agent_two(self) -> Agent:
        return Agent(
            config=self.agents_config['agent_two'],
            verbose=True
        )

    @task
```

```
    @task
    def task_two(self) -> Task:
        return Task(
            config=self.tasks_config['task_two']
        )

    @crew
    def crew(self) -> Crew:
        return Crew(
            agents=self.agents,  # Automatically collected by the @agent decorato
            tasks=self.tasks,    # Automatically collected by the @task decorator
            process=Process.sequential,
            verbose=True,
        )
```

Tasks will be executed in the order they are defined.

The `CrewBase` class, along with these decorators, automates the collection of agents and tasks, reducing the need for manual management.

Decorators overview from `annotations.py`

CrewAI provides several decorators in the `annotations.py` file that are used to mark methods within your crew class for special handling:

- `@CrewBase` : Marks the class as a crew base class.
- `@agent` : Denotes a method that returns an `Agent` object.
- `@task` : Denotes a method that returns a `Task` object.
- `@crew` : Denotes the method that returns the `Crew` object.
- `@before_kickoff` : (Optional) Marks a method to be executed before the crew starts.

Direct Code Definition (Alternative)

Alternatively, you can define the crew directly in code without using YAML configuration files.

code

```
from crewai import Agent, Crew, Task, Process
from crewai_tools import YourCustomTool

class YourCrewName:
    def agent_one(self) -> Agent:
        return Agent(
            role="Data Analyst",
            goal="Analyze data trends in the market",
            backstory="An experienced data analyst with a background in economics
            verbose=True,
            tools=[YourCustomTool()]
        )

    def agent_two(self) -> Agent:
        return Agent(
            role="Market Researcher",
            goal="Gather information on market dynamics",
            backstory="A diligent researcher with a keen eye for detail",
            verbose=True
        )

    def task_one(self) -> Task:
        return Task(
            description="Collect recent market data and identify trends.",
            expected_output="A report summarizing key trends in the market.",
            agent=self.agent_one()
        )
```

```
            agent=self.agent_two()
        )

    def crew(self) -> Crew:
        return Crew(
            agents=[self.agent_one(), self.agent_two()],
            tasks=[self.task_one(), self.task_two()],
            process=Process.sequential,
            verbose=True
        )
```

In this example:

- Agents and tasks are defined directly within the class without decorators.
- We manually create and manage the list of agents and tasks.
- This approach provides more control but can be less maintainable for larger projects.

Crew Output

The output of a crew in the CrewAI framework is encapsulated within the `CrewOutput` class. This class provides a structured way to access results of the crew's execution, including various formats such as raw strings, JSON, and Pydantic models. The `CrewOutput` includes the results from the final task output, token usage, and individual task outputs.

Crew Output Attributes

Attribute	Parameters	Type	Description
Raw	`raw`	`str`	The raw output of the crew. This is the default format for the output.

		`Any]]`	the crew.
Tasks Output	`tasks_output`	`List[TaskOutput]`	A list of `TaskOutput` objects, each representing the output of a task in the crew.
Token Usage	`token_usage`	`Dict[str, Any]`	A summary of token usage, providing insights into the language model's performance during execution.

Crew Output Methods and Properties

Method/Property	Description
json	Returns the JSON string representation of the crew output if the output format is JSON.
to_dict	Converts the JSON and Pydantic outputs to a dictionary.
str	Returns the string representation of the crew output, prioritizing Pydantic, then JSON, then raw.

Accessing Crew Outputs

Once a crew has been executed, its output can be accessed through the `output` attribute of the `Crew` object. The `CrewOutput` class provides various ways to interact with and present this output.

Example

Code

```
# Example crew execution
crew = Crew(
    agents=[research_agent, writer_agent],
    tasks=[research_task, write_article_task],
```

```
# Accessing the crew output
print(f"Raw Output: {crew_output.raw}")
if crew_output.json_dict:
    print(f"JSON Output: {json.dumps(crew_output.json_dict, indent=2)}")
if crew_output.pydantic:
    print(f"Pydantic Output: {crew_output.pydantic}")
print(f"Tasks Output: {crew_output.tasks_output}")
print(f"Token Usage: {crew_output.token_usage}")
```

Accessing Crew Logs

You can see real time log of the crew execution, by setting `output_log_file` as a `True(Boolean)` or a `file_name(str)`. Supports logging of events as both `file_name.txt` and `file_name.json`. In case of `True(Boolean)` will save as `logs.txt`.

In case of `output_log_file` is set as `False(Booelan)` or `None`, the logs will not be populated.

Code

```
# Save crew logs
crew = Crew(output_log_file = True)  # Logs will be saved as logs.txt
crew = Crew(output_log_file = file_name)  # Logs will be saved as file_name.txt
crew = Crew(output_log_file = file_name.txt)  # Logs will be saved as file_name.t
crew = Crew(output_log_file = file_name.json)  # Logs will be saved as file_name.
```

Memory Utilization

Crews can utilize memory (short-term, long-term, and entity memory) to enhance their execution and learning over time. This feature allows crews to store and recall execution

Caches can be employed to store the results of tools' execution, making the process more efficient by reducing the need to re-execute identical tasks.

Crew Usage Metrics

After the crew execution, you can access the `usage_metrics` attribute to view the language model (LLM) usage metrics for all tasks executed by the crew. This provides insights into operational efficiency and areas for improvement.

Code

```
# Access the crew's usage metrics
crew = Crew(agents=[agent1, agent2], tasks=[task1, task2])
crew.kickoff()
print(crew.usage_metrics)
```

Crew Execution Process

Sequential Process: Tasks are executed one after another, allowing for a linear flow of work.

Hierarchical Process: A manager agent coordinates the crew, delegating tasks and validating outcomes before proceeding. **Note**: A `manager_llm` or `manager_agent` is required for this process and it's essential for validating the process flow.

Kicking Off a Crew

Once your crew is assembled, initiate the workflow with the `kickoff()` method. This starts the execution process according to the defined process flow.

```
print(result)
```

Different Ways to Kick Off a Crew

Once your crew is assembled, initiate the workflow with the appropriate kickoff method. CrewAI provides several methods for better control over the kickoff process: `kickoff()`, `kickoff_for_each()`, `kickoff_async()`, and `kickoff_for_each_async()`.

- `kickoff()`: Starts the execution process according to the defined process flow.
- `kickoff_for_each()`: Executes tasks sequentially for each provided input event or item in the collection.
- `kickoff_async()`: Initiates the workflow asynchronously.
- `kickoff_for_each_async()`: Executes tasks concurrently for each provided input event or item, leveraging asynchronous processing.

Code

```
# Start the crew's task execution
result = my_crew.kickoff()
print(result)

# Example of using kickoff_for_each
inputs_array = [{'topic': 'AI in healthcare'}, {'topic': 'AI in finance'}]
results = my_crew.kickoff_for_each(inputs=inputs_array)
for result in results:
    print(result)

# Example of using kickoff_async
inputs = {'topic': 'AI in healthcare'}
async_result = my_crew.kickoff_async(inputs=inputs)
print(async_result)
```

```
print(async_result)
```

These methods provide flexibility in how you manage and execute tasks within your crew, allowing for both synchronous and asynchronous workflows tailored to your needs.

Replaying from a Specific Task

You can now replay from a specific task using our CLI command `replay` .

The replay feature in CrewAI allows you to replay from a specific task using the command-line interface (CLI). By running the command `crewai replay -t <task_id>` , you can specify the `task_id` for the replay process.

Kickoffs will now save the latest kickoffs returned task outputs locally for you to be able to replay from.

Replaying from a Specific Task Using the CLI

To use the replay feature, follow these steps:

1. Open your terminal or command prompt.
2. Navigate to the directory where your CrewAI project is located.
3. Run the following command:

To view the latest kickoff task IDs, use:

```
crewai log-tasks-outputs
```

Then, to replay from a specific task, use:

previously executed tasks.

‹ Tasks Flows ›

Core Concepts

Flows

Learn how to create and manage AI workflows using CrewAI Flows.

Introduction

CrewAI Flows is a powerful feature designed to streamline the creation and management of AI workflows. Flows allow developers to combine and coordinate coding tasks and Crews efficiently, providing a robust framework for building sophisticated AI automations.

Flows allow you to create structured, event-driven workflows. They provide a seamless way to connect multiple tasks, manage state, and control the flow of execution in your AI applications. With Flows, you can easily design and implement multi-step processes that leverage the full potential of CrewAI's capabilities.

1. **Simplified Workflow Creation**: Easily chain together multiple Crews and tasks to create complex AI workflows.
2. **State Management**: Flows make it super easy to manage and share state between different tasks in your workflow.
3. **Event-Driven Architecture**: Built on an event-driven model, allowing for dynamic and responsive workflows.
4. **Flexible Control Flow**: Implement conditional logic, loops, and branching within your workflows.

Getting Started

Core Concepts › Flows

```python
from crewai.flow.flow import Flow, listen, start
from dotenv import load_dotenv
from litellm import completion


class ExampleFlow(Flow):
    model = "gpt-4o-mini"

    @start()
    def generate_city(self):
        print("Starting flow")
        # Each flow state automatically gets a unique ID
        print(f"Flow State ID: {self.state['id']}")

        response = completion(
            model=self.model,
            messages=[
                {
                    "role": "user",
                    "content": "Return the name of a random city in the world.",
                },
            ],
        )

        random_city = response["choices"][0]["message"]["content"]
        # Store the city in our state
        self.state["city"] = random_city
        print(f"Random City: {random_city}")

        return random_city

    @listen(generate_city)
    def generate_fun_fact(self, random_city):
        response = completion(
```

```
                },
            ],
        )

        fun_fact = response["choices"][0]["message"]["content"]
        # Store the fun fact in our state
        self.state["fun_fact"] = fun_fact
        return fun_fact



flow = ExampleFlow()
result = flow.kickoff()

print(f"Generated fun fact: {result}")
```

In the above example, we have created a simple Flow that generates a random city using OpenAI and then generates a fun fact about that city. The Flow consists of two tasks: `generate_city` and `generate_fun_fact`. The `generate_city` task is the starting point of the Flow, and the `generate_fun_fact` task listens for the output of the `generate_city` task.

Each Flow instance automatically receives a unique identifier (UUID) in its state, which helps track and manage flow executions. The state can also store additional data (like the generated city and fun fact) that persists throughout the flow's execution.

When you run the Flow, it will:

1. Generate a unique ID for the flow state
2. Generate a random city and store it in the state
3. Generate a fun fact about that city and store it in the state

Note: Ensure you have set up your `.env` file to store your `OPENAI_API_KEY`. This key is necessary for authenticating requests to the OpenAI API.

@start()

The `@start()` decorator is used to mark a method as the starting point of a Flow. When a Flow is started, all the methods decorated with `@start()` are executed in parallel. You can have multiple start methods in a Flow, and they will all be executed when the Flow is started.

@listen()

The `@listen()` decorator is used to mark a method as a listener for the output of another task in the Flow. The method decorated with `@listen()` will be executed when the specified task emits an output. The method can access the output of the task it is listening to as an argument.

Usage

The `@listen()` decorator can be used in several ways:

1. **Listening to a Method by Name**: You can pass the name of the method you want to listen to as a string. When that method completes, the listener method will be triggered.

Code

```
@listen("generate_city")
def generate_fun_fact(self, random_city):
    # Implementation
```

```
@listen(generate_city)
def generate_fun_fact(self, random_city):
    # Implementation
```

Flow Output

Accessing and handling the output of a Flow is essential for integrating your AI workflows into larger applications or systems. CrewAI Flows provide straightforward mechanisms to retrieve the final output, access intermediate results, and manage the overall state of your Flow.

Retrieving the Final Output

When you run a Flow, the final output is determined by the last method that completes. The `kickoff()` method returns the output of this final method.

Here's how you can access the final output:

Code Output

```
from crewai.flow.flow import Flow, listen, start

class OutputExampleFlow(Flow):
    @start()
    def first_method(self):
        return "Output from first_method"

    @listen(first_method)
    def second_method(self, first_output):
        return f"Second method received: {first_output}"
```

In this example, the `second_method` is the last method to complete, so its output will be the final output of the Flow. The `kickoff()` method will return the final output, which is then printed to the console.

Accessing and Updating State

In addition to retrieving the final output, you can also access and update the state within your Flow. The state can be used to store and share data between different methods in the Flow. After the Flow has run, you can access the state to retrieve any information that was added or updated during the execution.

Here's an example of how to update and access the state:

Code Output

```
from crewai.flow.flow import Flow, listen, start
from pydantic import BaseModel

class ExampleState(BaseModel):
    counter: int = 0
    message: str = ""

class StateExampleFlow(Flow[ExampleState]):

    @start()
    def first_method(self):
        self.state.message = "Hello from first_method"
        self.state.counter += 1

    @listen(first_method)
    def second_method(self):
        self.state.message += " - updated by second_method"
```

```
print(f"Final Output: {final_output}")
print("Final State:")
print(flow.state)
```

In this example, the state is updated by both `first_method` and `second_method`. After the Flow has run, you can access the final state to see the updates made by these methods.

By ensuring that the final method's output is returned and providing access to the state, CrewAI Flows make it easy to integrate the results of your AI workflows into larger applications or systems, while also maintaining and accessing the state throughout the Flow's execution.

Flow State Management

Managing state effectively is crucial for building reliable and maintainable AI workflows. CrewAI Flows provides robust mechanisms for both unstructured and structured state management, allowing developers to choose the approach that best fits their application's needs.

Unstructured State Management

In unstructured state management, all state is stored in the `state` attribute of the `Flow` class. This approach offers flexibility, enabling developers to add or modify state attributes on the fly without defining a strict schema. Even with unstructured states, CrewAI Flows automatically generates and maintains a unique identifier (UUID) for each state instance.

Code

```
from crewai.flow.flow import Flow, listen, start
```

```
        print(f"State ID: {self.state['id']}")
        self.state['counter'] = 0
        self.state['message'] = "Hello from structured flow"

    @listen(first_method)
    def second_method(self):
        self.state['counter'] += 1
        self.state['message'] += " - updated"

    @listen(second_method)
    def third_method(self):
        self.state['counter'] += 1
        self.state['message'] += " - updated again"

        print(f"State after third_method: {self.state}")


flow = UnstructuredExampleFlow()
flow.kickoff()
```

Note: The `id` field is automatically generated and preserved throughout the flow's execution. You don't need to manage or set it manually, and it will be maintained even when updating the state with new data.

Key Points:

- **Flexibility:** You can dynamically add attributes to `self.state` without predefined constraints.
- **Simplicity:** Ideal for straightforward workflows where state structure is minimal or varies significantly.

Structured State Management

Each state in CrewAI Flows automatically receives a unique identifier (UUID) to help track and manage state instances. This ID is automatically generated and managed by the Flow system.

Code

```
from crewai.flow.flow import Flow, listen, start
from pydantic import BaseModel


class ExampleState(BaseModel):
    # Note: 'id' field is automatically added to all states
    counter: int = 0
    message: str = ""


class StructuredExampleFlow(Flow[ExampleState]):

    @start()
    def first_method(self):
        # Access the auto-generated ID if needed
        print(f"State ID: {self.state.id}")
        self.state.message = "Hello from structured flow"

    @listen(first_method)
    def second_method(self):
        self.state.counter += 1
        self.state.message += " - updated"

    @listen(second_method)
    def third_method(self):
        self.state.counter += 1
        self.state.message += " - updated again"
```

Key Points:

Defined Schema: `ExampleState` clearly outlines the state structure, enhancing code readability and maintainability.

Type Safety: Leveraging Pydantic ensures that state attributes adhere to the specified types, reducing runtime errors.

Auto-Completion: IDEs can provide better auto-completion and error checking based on the defined state model.

Choosing Between Unstructured and Structured State Management

Use Unstructured State Management when:

The workflow's state is simple or highly dynamic.

Flexibility is prioritized over strict state definitions.

Rapid prototyping is required without the overhead of defining schemas.

Use Structured State Management when:

The workflow requires a well-defined and consistent state structure.

Type safety and validation are important for your application's reliability.

You want to leverage IDE features like auto-completion and type checking for better developer experience.

By providing both unstructured and structured state management options, CrewAI Flows empowers developers to build AI workflows that are both flexible and robust, catering to a wide range of application requirements.

be applied at either the class level or method level, providing flexibility in how you manage state persistence.

Class-Level Persistence

When applied at the class level, the @persist decorator automatically persists all flow method states:

```
@persist  # Using SQLiteFlowPersistence by default
class MyFlow(Flow[MyState]):
    @start()
    def initialize_flow(self):
        # This method will automatically have its state persisted
        self.state.counter = 1
        print("Initialized flow. State ID:", self.state.id)

    @listen(initialize_flow)
    def next_step(self):
        # The state (including self.state.id) is automatically reloaded
        self.state.counter += 1
        print("Flow state is persisted. Counter:", self.state.counter)
```

Method-Level Persistence

For more granular control, you can apply @persist to specific methods:

```
class AnotherFlow(Flow[dict]):
    @persist  # Persists only this method's state
    @start()
    def begin(self):
        if "runs" not in self.state:
            self.state["runs"] = 0
```

How It Works

1. **Unique State Identification**

   Each flow state automatically receives a unique UUID

   The ID is preserved across state updates and method calls

   Supports both structured (Pydantic BaseModel) and unstructured (dictionary) states

2. **Default SQLite Backend**

   SQLiteFlowPersistence is the default storage backend

   States are automatically saved to a local SQLite database

   Robust error handling ensures clear messages if database operations fail

3. **Error Handling**

   Comprehensive error messages for database operations

   Automatic state validation during save and load

   Clear feedback when persistence operations encounter issues

Important Considerations

State Types: Both structured (Pydantic BaseModel) and unstructured (dictionary) states are supported

Automatic ID: The `id` field is automatically added if not present

State Recovery: Failed or restarted flows can automatically reload their previous state

Custom Implementation: You can provide your own FlowPersistence implementation for specialized storage needs

- Fine-grained control via method-level persistence decorators
- Built-in state inspection and debugging capabilities
- Full visibility into state changes and persistence operations

2. **Enhanced Reliability**
   - Automatic state recovery after system failures or restarts
   - Transaction-based state updates for data integrity
   - Comprehensive error handling with clear error messages
   - Robust validation during state save and load operations
3. **Extensible Architecture**
   - Customizable persistence backend through FlowPersistence interface
   - Support for specialized storage solutions beyond SQLite
   - Compatible with both structured (Pydantic) and unstructured (dict) states
   - Seamless integration with existing CrewAI flow patterns

The persistence system's architecture emphasizes technical precision and customization options, allowing developers to maintain full control over state management while benefiting from built-in reliability features.

Flow Control

Conditional Logic: `or`

The `or_` function in Flows allows you to listen to multiple methods and trigger the listener method when any of the specified methods emit an output.

```
class OrExampleFlow(Flow):

    @start()
    def start_method(self):
        return "Hello from the start method"

    @listen(start_method)
    def second_method(self):
        return "Hello from the second method"

    @listen(or_(start_method, second_method))
    def logger(self, result):
        print(f"Logger: {result}")



flow = OrExampleFlow()
flow.kickoff()
```

When you run this Flow, the `logger` method will be triggered by the output of either the `start_method` or the `second_method`. The `or_` function is used to listen to multiple methods and trigger the listener method when any of the specified methods emit an output.

Conditional Logic: and

The `and_` function in Flows allows you to listen to multiple methods and trigger the listener method only when all the specified methods emit an output.

Code Output

```
from crewai.flow.flow import Flow, and_, listen, start

class AndExampleFlow(Flow):
```

```
    @listen(start_method)
    def second_method(self):
        self.state["joke"] = "What do computers eat? Microchips."

    @listen(and_(start_method, second_method))
    def logger(self):
        print("---- Logger ----")
        print(self.state)

flow = AndExampleFlow()
flow.kickoff()
```

When you run this Flow, the `logger` method will be triggered only when both the `start_method` and the `second_method` emit an output. The `and_` function is used to listen to multiple methods and trigger the listener method only when all the specified methods emit an output.

Router

The `@router()` decorator in Flows allows you to define conditional routing logic based on the output of a method. You can specify different routes based on the output of the method, allowing you to control the flow of execution dynamically.

Code Output

```
import random
from crewai.flow.flow import Flow, listen, router, start
from pydantic import BaseModel

class ExampleState(BaseModel):
    success_flag: bool = False
```

```
        random_boolean = random.choice([True, False])
        self.state.success_flag = random_boolean

    @router(start_method)
    def second_method(self):
        if self.state.success_flag:
            return "success"
        else:
            return "failed"

    @listen("success")
    def third_method(self):
        print("Third method running")

    @listen("failed")
    def fourth_method(self):
        print("Fourth method running")


flow = RouterFlow()
flow.kickoff()
```

In the above example, the `start_method` generates a random boolean value and sets it in the state. The `second_method` uses the `@router()` decorator to define conditional routing logic based on the value of the boolean. If the boolean is `True`, the method returns `"success"`, and if it is `False`, the method returns `"failed"`. The `third_method` and `fourth_method` listen to the output of the `second_method` and execute based on the returned value.

When you run this Flow, the output will change based on the random boolean value generated by the `start_method`.

flow with multiple crews by running the following command:

```
crewai create flow name_of_flow
```

This command will generate a new CrewAI project with the necessary folder structure. The generated project includes a prebuilt crew called `poem_crew` that is already working. You can use this crew as a template by copying, pasting, and editing it to create other crews.

Folder Structure

After running the `crewai create flow name_of_flow` command, you will see a folder structure similar to the following:

Directory/File	Description
`name_of_flow/`	Root directory for the flow.
`├── crews/`	Contains directories for specific crews.
`│ └── poem_crew/`	Directory for the “poem_crew” with its configurations and scripts.
`│ ├── config/`	Configuration files directory for the “poem_crew”.
`│ │ ├── agents.yaml`	YAML file defining the agents for “poem_crew”.
`│ │ └── tasks.yaml`	YAML file defining the tasks for “poem_crew”.
`│ ├── poem_crew.py`	Script for “poem_crew” functionality.
`├── tools/`	Directory for additional tools used in the flow.
`│ └── custom_tool.py`	Custom tool implementation.
`├── main.py`	Main script for running the flow.
`├── README.md`	Project description and instructions.

Core Concepts › Flows

Building Your Crews

In the `crews` folder, you can define multiple crews. Each crew will have its own folder containing configuration files and the crew definition file. For example, the `poem_crew` folder contains:

- `config/agents.yaml` : Defines the agents for the crew.
- `config/tasks.yaml` : Defines the tasks for the crew.
- `poem_crew.py` : Contains the crew definition, including agents, tasks, and the crew itself.

You can copy, paste, and edit the `poem_crew` to create other crews.

Connecting Crews in `main.py`

The `main.py` file is where you create your flow and connect the crews together. You can define your flow by using the `Flow` class and the decorators `@start` and `@listen` to specify the flow of execution.

Here's an example of how you can connect the `poem_crew` in the `main.py` file:

Code

```
#!/usr/bin/env python
from random import randint

from pydantic import BaseModel
from crewai.flow.flow import Flow, listen, start
from .crews.poem_crew.poem_crew import PoemCrew

class PoemState(BaseModel):
```

```
    @start()
    def generate_sentence_count(self):
        print("Generating sentence count")
        self.state.sentence_count = randint(1, 5)

    @listen(generate_sentence_count)
    def generate_poem(self):
        print("Generating poem")
        result = PoemCrew().crew().kickoff(inputs={"sentence_count": self.state.s

        print("Poem generated", result.raw)
        self.state.poem = result.raw

    @listen(generate_poem)
    def save_poem(self):
        print("Saving poem")
        with open("poem.txt", "w") as f:
            f.write(self.state.poem)

def kickoff():
    poem_flow = PoemFlow()
    poem_flow.kickoff()


def plot():
    poem_flow = PoemFlow()
    poem_flow.plot()

if __name__ == "__main__":
    kickoff()
```

In this example, the `PoemFlow` class defines a flow that generates a sentence count, uses the `PoemCrew` to generate a poem, and then saves the poem to a file. The flow is kicked off

(Optional) Before running the flow, you can install the dependencies by running:

```
crewai install
```

Once all of the dependencies are installed, you need to activate the virtual environment by running:

```
source .venv/bin/activate
```

After activating the virtual environment, you can run the flow by executing one of the following commands:

```
crewai flow kickoff
```

or

```
uv run kickoff
```

The flow will execute, and you should see the output in the console.

Plot Flows

Visualizing your AI workflows can provide valuable insights into the structure and execution paths of your flows. CrewAI offers a powerful visualization tool that allows you to generate

What are Plots?

Plots in CrewAI are graphical representations of your AI workflows. They display the various tasks, their connections, and the flow of data between them. This visualization helps in understanding the sequence of operations, identifying bottlenecks, and ensuring that the workflow logic aligns with your expectations.

How to Generate a Plot

CrewAI provides two convenient methods to generate plots of your flows:

Option 1: Using the `plot()` Method

If you are working directly with a flow instance, you can generate a plot by calling the `plot()` method on your flow object. This method will create an HTML file containing the interactive plot of your flow.

Code

```
# Assuming you have a flow instance
flow.plot("my_flow_plot")
```

This will generate a file named `my_flow_plot.html` in your current directory. You can open this file in a web browser to view the interactive plot.

Option 2: Using the Command Line

If you are working within a structured CrewAI project, you can generate a plot using the command line. This is particularly useful for larger projects where you want to visualize the entire flow setup.

method. The file will be saved in your project directory, and you can open it in a web browser to explore the flow.

Understanding the Plot

The generated plot will display nodes representing the tasks in your flow, with directed edges indicating the flow of execution. The plot is interactive, allowing you to zoom in and out, and hover over nodes to see additional details.

By visualizing your flows, you can gain a clearer understanding of the workflow's structure, making it easier to debug, optimize, and communicate your AI processes to others.

Conclusion

Plotting your flows is a powerful feature of CrewAI that enhances your ability to design and manage complex AI workflows. Whether you choose to use the `plot()` method or the command line, generating plots will provide you with a visual representation of your workflows, aiding in both development and presentation.

Next Steps

If you're interested in exploring additional examples of flows, we have a variety of recommendations in our examples repository. Here are four specific flow examples, each showcasing unique use cases to help you match your current problem type to a specific example:

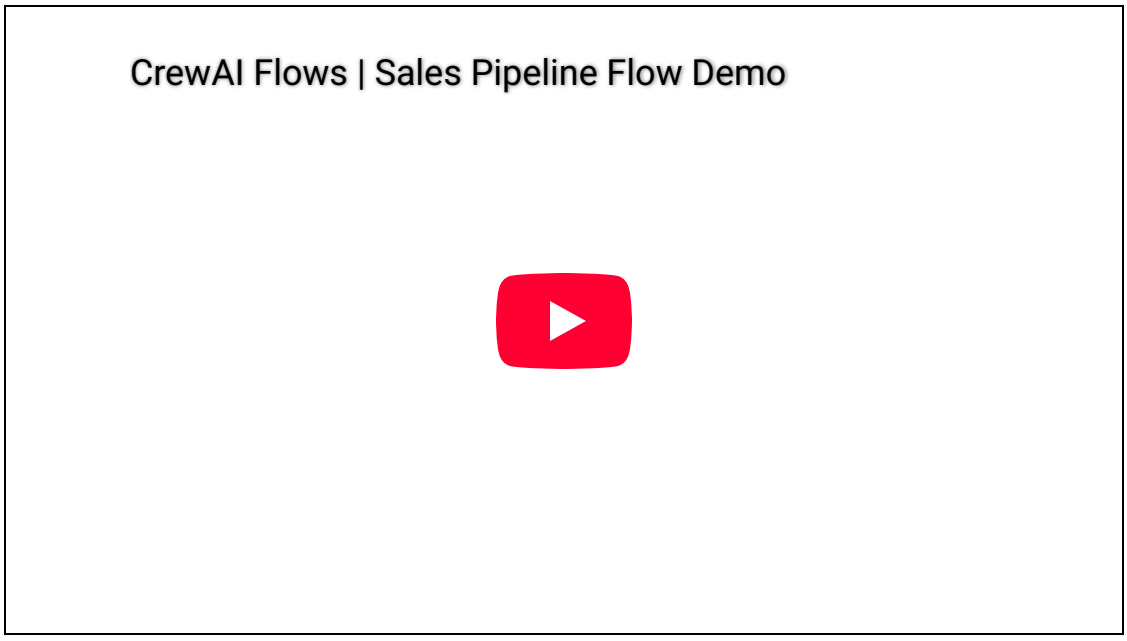
1. **Email Auto Responder Flow**: This example demonstrates an infinite loop where a background job continually runs to automate email responses. It's a great use case for tasks that need to be performed repeatedly without manual intervention. View Example

3. **Write a Book Flow**: This example excels at chaining multiple crews together, where the output of one crew is used by another. Specifically, one crew outlines an entire book, and another crew generates chapters based on the outline. Eventually, everything is connected to produce a complete book. This flow is perfect for complex, multi-step processes that require coordination between different tasks. View Example

4. **Meeting Assistant Flow**: This flow demonstrates how to broadcast one event to trigger multiple follow-up actions. For instance, after a meeting is completed, the flow can update a Trello board, send a Slack message, and save the results. It's a great example of handling multiple outcomes from a single event, making it ideal for comprehensive task management and notification systems. View Example

By exploring these examples, you can gain insights into how to leverage CrewAI Flows for various use cases, from automating repetitive tasks to managing complex, multi-step processes with dynamic decision-making and human feedback.

Also, check out our YouTube video on how to use flows in CrewAI below!



Core Concepts

Knowledge

What is knowledge in CrewAI and how to use it.

What is Knowledge?

Knowledge in CrewAI is a powerful system that allows AI agents to access and utilize external information sources during their tasks. Think of it as giving your agents a reference library they can consult while working.

Key benefits of using Knowledge:

- Enhance agents with domain-specific information
- Support decisions with real-world data
- Maintain context across conversations
- Ground responses in factual information

Supported Knowledge Sources

CrewAI supports various types of knowledge sources out of the box:

Text Sources

Raw strings
Text files (.txt)
PDF documents

Structured Data

CSV files
Excel spreadsheets
JSON documents

			include PDF, CSV, Excel, JSON, text files, or string content.
`collection_name`	str	No	Name of the collection where the knowledge will be stored. Used to identify different sets of knowledge. Defaults to “knowledge” if not provided.
`storage`	Optional[KnowledgeStorage]	No	Custom storage configuration for managing how the knowledge is stored and retrieved. If not provided, a default storage will be created.

Quickstart Example

For file-Based Knowledge Sources, make sure to place your files in a `knowledge` directory at the root of your project. Also, use relative paths from the `knowledge` directory when creating the source.

Here’s an example using string-based knowledge:

Code

```
from crewai import Agent, Task, Crew, Process, LLM
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSource

# Create a knowledge source
content = "Users name is John. He is 30 years old and lives in San Francisco."
string_source = StringKnowledgeSource(
    content=content,
)

# Create an LLM with a temperature of 0 to ensure deterministic outputs
llm = LLM(model="gpt-4o-mini", temperature=0)
```

```
    verbose=True,
    allow_delegation=False,
    llm=llm,
)
task = Task(
    description="Answer the following questions about the user: {question}",
    expected_output="An answer to the question.",
    agent=agent,
)

crew = Crew(
    agents=[agent],
    tasks=[task],
    verbose=True,
    process=Process.sequential,
    knowledge_sources=[string_source], # Enable knowledge by adding the sources h
)

result = crew.kickoff(inputs={"question": "What city does John live in and how ol
```

Here's another example with the `CrewDoclingSource`. The CrewDoclingSource is actually quite versatile and can handle multiple file formats including MD, PDF, DOCX, HTML, and more.

> You need to install `docling` for the following example to work: `uv add docling`

Code

```
from crewai import LLM, Agent, Crew, Process, Task
from crewai.knowledge.source.crew_docling_source import CrewDoclingSource

# Create a knowledge source
```

```
)

# Create an LLM with a temperature of 0 to ensure deterministic outputs
llm = LLM(model="gpt-4o-mini", temperature=0)

# Create an agent with the knowledge store
agent = Agent(
    role="About papers",
    goal="You know everything about the papers.",
    backstory="""You are a master at understanding papers and their content.""",
    verbose=True,
    allow_delegation=False,
    llm=llm,
)
task = Task(
    description="Answer the following questions about the papers: {question}",
    expected_output="An answer to the question.",
    agent=agent,
)

crew = Crew(
    agents=[agent],
    tasks=[task],
    verbose=True,
    process=Process.sequential,
    knowledge_sources=[
        content_source
    ],  # Enable knowledge by adding the sources here. You can also add more sour
)

result = crew.kickoff(
    inputs={
        "question": "What is the reward hacking paper about? Be sure to provide s
    }
)
```

crewai

Core Concepts › Knowledge

Here are examples of how to use different types of knowledge sources:

Text File Knowledge Source

```
from crewai.knowledge.source.text_file_knowledge_source import TextFileKnowle  jeS

# Create a text file knowledge source
text_source = TextFileKnowledgeSource(
    file_paths=["document.txt", "another.txt"]
)

# Create crew with text file source on agents or crew level
agent = Agent(
    ...
    knowledge_sources=[text_source]
)

crew = Crew(
    ...
    knowledge_sources=[text_source]
)
```

PDF Knowledge Source

```
from crewai.knowledge.source.pdf_knowledge_source import PDFKnowledgeSource

# Create a PDF knowledge source
pdf_source = PDFKnowledgeSource(
    file_paths=["document.pdf", "another.pdf"]
)
```

```
)

crew = Crew(
    ...
    knowledge_sources=[pdf_source]
)
```

CSV Knowledge Source

```
from crewai.knowledge.source.csv_knowledge_source import CSVKnowledgeSource

# Create a CSV knowledge source
csv_source = CSVKnowledgeSource(
    file_paths=["data.csv"]
)

# Create crew with CSV knowledge source or on agent level
agent = Agent(
    ...
    knowledge_sources=[csv_source]
)

crew = Crew(
    ...
    knowledge_sources=[csv_source]
)
```

Excel Knowledge Source

```
    file_paths=["spreadsheet.xlsx"]
)

# Create crew with Excel knowledge source on agents or crew level
agent = Agent(
    ...
    knowledge_sources=[excel_source]
)

crew = Crew(
    ...
    knowledge_sources=[excel_source]
)
```

JSON Knowledge Source

```
from crewai.knowledge.source.json_knowledge_source import JSONKnowledgeSource

# Create a JSON knowledge source
json_source = JSONKnowledgeSource(
    file_paths=["data.json"]
)

# Create crew with JSON knowledge source on agents or crew level
agent = Agent(
    ...
    knowledge_sources=[json_source]
)

crew = Crew(
    ...
```

Core Concepts › Knowledge

Knowledge Configuration

Chunking Configuration

Knowledge sources automatically chunk content for better processing. You can configure chunking behavior in your knowledge sources:

```
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSc ·ce

source = StringKnowledgeSource(
    content="Your content here",
    chunk_size=4000,      # Maximum size of each chunk (default: 4000)
    chunk_overlap=200     # Overlap between chunks (default: 200)
)
```

The chunking configuration helps in:

- Breaking down large documents into manageable pieces
- Maintaining context through chunk overlap
- Optimizing retrieval accuracy

Embeddings Configuration

You can also configure the embedder for the knowledge store. This is useful if you want to use a different embedder for the knowledge store than the one used for the agents. The `embedder` parameter supports various embedding model providers that include:

- `openai` : OpenAI's embedding models
- `google` : Google's text embedding models
- `azure` : Azure OpenAI embeddings

Core Concepts › Knowledge

`voyageai` : VoyageAI's embedding models

`bedrock` : AWS Bedrock embeddings

`huggingface` : Hugging Face models

`watson` : IBM Watson embeddings

Here's an example of how to configure the embedder for the knowledge store using Google's `text-embedding-004` model:

Example Output

```
from crewai import Agent, Task, Crew, Process, LLM
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSourc
import os

# Get the GEMINI API key
GEMINI_API_KEY = os.environ.get("GEMINI_API_KEY")

# Create a knowledge source
content = "Users name is John. He is 30 years old and lives in San Francisco."
string_source = StringKnowledgeSource(
    content=content,
)

# Create an LLM with a temperature of 0 to ensure deterministic outputs
gemini_llm = LLM(
    model="gemini/gemini-1.5-pro-002",
    api_key=GEMINI_API_KEY,
    temperature=0,
)

# Create an agent with the knowledge store
agent = Agent(
    role="About User",
    goal="You know everything about the user.",
```

```
        "provider": "google",
        "config": {
            "model": "models/text-embedding-004",
            "api_key": GEMINI_API_KEY,
        }
    }
)

task = Task(
    description="Answer the following questions about the user: {question}",
    expected_output="An answer to the question.",
    agent=agent,
)

crew = Crew(
    agents=[agent],
    tasks=[task],
    verbose=True,
    process=Process.sequential,
    knowledge_sources=[string_source],
    embedder={
        "provider": "google",
        "config": {
            "model": "models/text-embedding-004",
            "api_key": GEMINI_API_KEY,
        }
    }
)

result = crew.kickoff(inputs={"question": "What city does John live in and how o
```

Clearing Knowledge

```
crewai reset-memories --knowledge
```

This is useful when you've updated your knowledge sources and want to ensure that the agents are using the most recent information.

Agent-Specific Knowledge

While knowledge can be provided at the crew level using `crew.knowledge_sources`, individual agents can also have their own knowledge sources using the `knowledge_sources` parameter:

Code

```
from crewai import Agent, Task, Crew
from crewai.knowledge.source.string_knowledge_source import StringKnowledgeSource

# Create agent-specific knowledge about a product
product_specs = StringKnowledgeSource(
    content="""The XPS 13 laptop features:
    - 13.4-inch 4K display
    - Intel Core i7 processor
    - 16GB RAM
    - 512GB SSD storage
    - 12-hour battery life""",
    metadata={"category": "product_specs"}
)

# Create a support agent with product knowledge
support_agent = Agent(
    role="Technical Support Specialist",
    goal="Provide accurate product information and support.",
```

```
support_task = Task(
    description="Answer this customer question: {question}",
    agent=support_agent
)

# Create and run the crew
crew = Crew(
    agents=[support_agent],
    tasks=[support_task]
)

# Get answer about the laptop's specifications
result = crew.kickoff(
    inputs={"question": "What is the storage capacity of the XPS 13?"}
)
```

Benefits of agent-specific knowledge:

- Give agents specialized information for their roles
- Maintain separation of concerns between agents
- Combine with crew-level knowledge for layered information access

Custom Knowledge Sources

CrewAI allows you to create custom knowledge sources for any type of data by extending the `BaseKnowledgeSource` class. Let's create a practical example that fetches and processes space news articles.

Space News Knowledge Source Example

```python
import requests
from datetime import datetime
from typing import Dict, Any
from pydantic import BaseModel, Field

class SpaceNewsKnowledgeSource(BaseKnowledgeSource):
    """Knowledge source that fetches data from Space News API."""

    api_endpoint: str = Field(description="API endpoint URL")
    limit: int = Field(default=10, description="Number of articles to fetch")

    def load_content(self) -> Dict[Any, str]:
        """Fetch and format space news articles."""
        try:
            response = requests.get(
                f"{self.api_endpoint}?limit={self.limit}"
            )
            response.raise_for_status()

            data = response.json()
            articles = data.get('results', [])

            formatted_data = self._format_articles(articles)
            return {self.api_endpoint: formatted_data}
        except Exception as e:
            raise ValueError(f"Failed to fetch space news: {str(e)}")

    def _format_articles(self, articles: list) -> str:
        """Format articles into readable text."""
        formatted = "Space News Articles:\n\n"
        for article in articles:
            formatted += f"""
                Title: {article['title']}
                Published: {article['published_at']}
                Summary: {article['summary']}
                News Site: {article['news_site']}
```

```
        """Process and store the articles."""
        content = self.load_content()
        for _, text in content.items():
            chunks = self._chunk_text(text)
            self.chunks.extend(chunks)

        self._save_documents()

# Create knowledge source
recent_news = SpaceNewsKnowledgeSource(
    api_endpoint="https://api.spaceflightnewsapi.net/v4/articles",
    limit=10,
)

# Create specialized agent
space_analyst = Agent(
    role="Space News Analyst",
    goal="Answer questions about space news accurately and comprehensively",
    backstory="""You are a space industry analyst with expertise in space explor
    satellite technology, and space industry trends. You excel at answering ques
    about space news and providing detailed, accurate information.""",
    knowledge_sources=[recent_news],
    llm=LLM(model="gpt-4", temperature=0.0)
)

# Create task that handles user questions
analysis_task = Task(
    description="Answer this question about space news: {user_question}",
    expected_output="A detailed answer based on the recent space news articles",
    agent=space_analyst
)

# Create and run the crew
crew = Crew(
    agents=[space_analyst],
```

```
# Example usage
result = crew.kickoff(
    inputs={"user_question": "What are the latest developments in space explorat
)
```

Key Components Explained

1. **Custom Knowledge Source (`SpaceNewsKnowledgeSource`)**:

   Extends `BaseKnowledgeSource` for integration with CrewAI

   Configurable API endpoint and article limit

   Implements three key methods:

   `load_content()` : Fetches articles from the API

   `_format_articles()` : Structures the articles into readable text

   `add()` : Processes and stores the content

2. **Agent Configuration**:

   Specialized role as a Space News Analyst

   Uses the knowledge source to access space news

3. **Task Setup**:

   Takes a user question as input through `{user_question}`

   Designed to provide detailed answers based on the knowledge source

4. **Crew Orchestration**:

   Manages the workflow between agent and task

   Handles input/output through the kickoff method

Use the knowledge source to answer specific user questions

Integrate everything seamlessly with CrewAI's agent system

About the Spaceflight News API

The example uses the Spaceflight News API, which:

Provides free access to space-related news articles

Requires no authentication

Returns structured data about space news

Supports pagination and filtering

You can customize the API query by modifying the endpoint URL:

```
# Fetch more articles
recent_news = SpaceNewsKnowledgeSource(
    api_endpoint="https://api.spaceflightnewsapi.net/v4/articles",
    limit=20,  # Increase the number of articles
)

# Add search parameters
recent_news = SpaceNewsKnowledgeSource(
    api_endpoint="https://api.spaceflightnewsapi.net/v4/articles?search=NASA", #
    limit=10,
)
```

Best Practices

Content Organization

Was this page helpful? Yes No

‹ Flows LLMs ›

Core Concepts › LLMs

Core Concepts

LLMs

A comprehensive guide to configuring and using Large Language Models (LLMs) in your CrewAI projects

CrewAI integrates with multiple LLM providers through LiteLLM, giving you the flexibility to choose the right model for your specific use case. This guide will help you understand how to configure and use different LLM providers in your CrewAI projects.

What are LLMs?

Large Language Models (LLMs) are the core intelligence behind CrewAI agents. They enable agents to understand context, make decisions, and generate human-like responses. Here's what you need to know:

LLM Basics

Large Language Models are AI systems trained on vast amounts of text data. They power the intelligence of your CrewAI agents, enabling them to understand and generate human-like text.

Context Window

The context window determines how much text an LLM can process at once. Larger windows (e.g., 128K tokens) allow for more context but may be more expensive and slower.

Temperature

Provider Selection

Core Concepts › LLMs

Setting Up Your LLM

There are three ways to configure LLMs in CrewAI. Choose the method that best fits your workflow:

1. Environment Variables | 2. YAML Configuration | 3. Direct Code

The simplest way to get started. Set these variables in your environment:

```
# Required: Your API key for authentication
OPENAI_API_KEY=<your-api-key>

# Optional: Default model selection
OPENAI_MODEL_NAME=gpt-4o-mini  # Default if not set

# Optional: Organization ID (if applicable)
OPENAI_ORGANIZATION_ID=<your-org-id>
```

⚠ Never commit API keys to version control. Use environment files (.env) or your system's secret management.

Provider Configuration Examples

CrewAI supports a multitude of LLM providers, each offering unique features, authentication methods, and model capabilities. In this section, you'll find detailed examples that help you select, configure, and optimize the LLM that best fits your project's needs.

Google
Azure
AWS Bedrock
Amazon SageMaker
Mistral
Nvidia NIM
Groq
IBM watsonx.ai
Ollama (Local LLMs)
Fireworks AI
Perplexity AI
Hugging Face
SambaNova
Cerebras
Open Router

Structured LLM Calls

For example, you can define a Pydantic model to represent the expected response structure and pass it as the `response_format` when instantiating the LLM. The model will then be used to convert the LLM output into a structured Python object.

Code

```
from crewai import LLM

class Dog(BaseModel):
    name: str
    age: int
    breed: str


llm = LLM(model="gpt-4o", response_format=Dog)

response = llm.call(
    "Analyze the following messages and return the name, age, and breed. "
    "Meet Kona! She is 3 years old and is a black german shepherd."
)
print(response)

# Output:
# Dog(name='Kona', age=3, breed='black german shepherd')
```

Advanced Features and Optimization

Learn how to get the most out of your LLM configuration:

Context Window Management

Core Concepts › LLMs

Common issues and solutions

Authentication | Model Names | Context Length

⚠ Most authentication issues can be resolved by checking API key format and environment variable names.

```
# OpenAI
OPENAI_API_KEY=sk-...

# Anthropic
ANTHROPIC_API_KEY=sk-ant-...
```

Getting Help

If you need assistance, these resources are available:

LiteLLM Documentation
Comprehensive documentation for LiteLLM integration and troubleshooting common issues.

GitHub Issues
Report bugs, request features, or browse existing issues for solutions.

Community Forum
Connect with other CrewAI users, share experiences, and get help from the community.

Rotate keys regularly

Use separate keys for development and production

Monitor key usage for unusual patterns

Was this page helpful? Yes No

‹ Knowledge Processes ›

Core Concepts

Processes

Detailed guide on workflow management through processes in CrewAI, with updated implementation details.

Understanding Processes

Processes orchestrate the execution of tasks by agents, akin to project management in human teams. These processes ensure tasks are distributed and executed efficiently, in alignment with a predefined strategy.

Process Implementations

Sequential: Executes tasks sequentially, ensuring tasks are completed in an orderly progression.

Hierarchical: Organizes tasks in a managerial hierarchy, where tasks are delegated and executed based on a structured chain of command. A manager language model (`manager_llm`) or a custom manager agent (`manager_agent`) must be specified in the crew to enable the hierarchical process, facilitating the creation and management of tasks by the manager.

Consensual Process (Planned): Aiming for collaborative decision-making among agents on task execution, this process type introduces a democratic approach to task management within CrewAI. It is planned for future development and is not currently implemented in the codebase.

The Role of Processes in Teamwork

Assigning Processes to a Crew

To assign a process to a crew, specify the process type upon crew creation to set the execution strategy. For a hierarchical process, ensure to define `manager_llm` or `manager_agent` for the manager agent.

```
from crewai import Crew, Process

# Example: Creating a crew with a sequential process
crew = Crew(
    agents=my_agents,
    tasks=my_tasks,
    process=Process.sequential
)

# Example: Creating a crew with a hierarchical process
# Ensure to provide a manager_llm or manager_agent
crew = Crew(
    agents=my_agents,
    tasks=my_tasks,
    process=Process.hierarchical,
    manager_llm="gpt-4o"
    # or
    # manager_agent=my_manager_agent
)
```

Note: Ensure `my_agents` and `my_tasks` are defined prior to creating a `Crew` object, and for the hierarchical process, either `manager_llm` or `manager_agent` is also required.

Sequential Process

To customize task context, utilize the `context` parameter in the `Task` class to specify outputs that should be used as context for subsequent tasks.

Hierarchical Process

Emulates a corporate hierarchy, CrewAI allows specifying a custom manager agent or automatically creates one, requiring the specification of a manager language model (`manager_llm`). This agent oversees task execution, including planning, delegation, and validation. Tasks are not pre-assigned; the manager allocates tasks to agents based on their capabilities, reviews outputs, and assesses task completion.

Process Class: Detailed Overview

The `Process` class is implemented as an enumeration (`Enum`), ensuring type safety and restricting process values to the defined types (`sequential` , `hierarchical`). The consensual process is planned for future inclusion, emphasizing our commitment to continuous development and innovation.

Conclusion

The structured collaboration facilitated by processes within CrewAI is crucial for enabling systematic teamwork among agents. This documentation has been updated to reflect the latest features, enhancements, and the planned integration of the Consensual Process, ensuring users have access to the most current and comprehensive information.

Core Concepts

Collaboration

Exploring the dynamics of agent collaboration within the CrewAI framework, focusing on the newly integrated features for enhanced functionality.

Collaboration Fundamentals

Collaboration in CrewAI is fundamental, enabling agents to combine their skills, share information, and assist each other in task execution, embodying a truly cooperative ecosystem.

- **Information Sharing**: Ensures all agents are well-informed and can contribute effectively by sharing data and findings.
- **Task Assistance**: Allows agents to seek help from peers with the required expertise for specific tasks.
- **Resource Allocation**: Optimizes task execution through the efficient distribution and sharing of resources among agents.

Enhanced Attributes for Improved Collaboration

The `Crew` class has been enriched with several attributes to support advanced functionalities:

Feature	Description
Language Model Management (`manager_llm`, `function_calling_llm`)	Manages language models for executing tasks and tools. `manager_llm` is required for hierarchical processes, while `function_calling_llm` is optional with a default value for streamlined interactions.

	distribution.
Verbose Logging (`verbose`)	Provides detailed logging for monitoring and debugging. Accepts integer and boolean values to control verbosity level.
Rate Limiting (`max_rpm`)	Limits requests per minute to optimize resource usage. Setting guidelines depend on task complexity and load.
Internationalization / Customization (`language` , `prompt_file`)	Supports prompt customization for global usability. Example of file
Execution and Output Handling (`full_output`)	Controls output granularity, distinguishing between full and final outputs.
Callback and Telemetry (`step_callback` , `task_callback`)	Enables step-wise and task-level execution monitoring and telemetry for performance analytics.
Crew Sharing (`share_crew`)	Allows sharing crew data with CrewAI for model improvement. Privacy implications and benefits should be considered.
Usage Metrics (`usage_metrics`)	Logs all LLM usage metrics during task execution for performance insights.
Memory Usage (`memory`)	Enables memory for storing execution history, aiding in agent learning and task efficiency.
Embedder Configuration (`embedder`)	Configures the embedder for language understanding and generation, with support for provider customization.
Cache Management (`cache`)	Specifies whether to cache tool execution results, enhancing performance.
Output Logging (`output_log_file`)	Defines the file path for logging crew execution output.
Planning Mode (`planning`)	Enables action planning before task execution. Set `planning=True` to activate.
Replay Feature (`replay`)	Provides CLI for listing tasks from the last run and replaying from specific tasks, aiding in task management and troubleshooting.

Delegation (Dividing to Conquer)

Implementing Collaboration and Delegation

Setting up a crew involves defining the roles and capabilities of each agent. CrewAI seamlessly manages their interactions, ensuring efficient collaboration and delegation, with enhanced customization and monitoring features to adapt to various operational needs.

Example Scenario

Consider a crew with a researcher agent tasked with data gathering and a writer agent responsible for compiling reports. The integration of advanced language model management and process flow attributes allows for more sophisticated interactions, such as the writer delegating complex research tasks to the researcher or querying specific information, thereby facilitating a seamless workflow.

Conclusion

The integration of advanced attributes and functionalities into the CrewAI framework significantly enriches the agent collaboration ecosystem. These enhancements not only simplify interactions but also offer unprecedented flexibility and control, paving the way for sophisticated AI-driven solutions capable of tackling complex tasks through intelligent collaboration and delegation.

‹ Processes Training ›

Core Concepts

Training

Learn how to train your CrewAI agents by giving them feedback early on and get consistent results.

Introduction

The training feature in CrewAI allows you to train your AI agents using the command-line interface (CLI). By running the command `crewai train -n <n_iterations>`, you can specify the number of iterations for the training process.

During training, CrewAI utilizes techniques to optimize the performance of your agents along with human feedback. This helps the agents improve their understanding, decision-making, and problem-solving abilities.

Training Your Crew Using the CLI

To use the training feature, follow these steps:

1. Open your terminal or command prompt.
2. Navigate to the directory where your CrewAI project is located.
3. Run the following command:

```
crewai train -n <n_iterations> <filename> (optional)
```

Core Concepts › Training

Training Your Crew Programmatically

To train your crew programmatically, use the following steps:

1. Define the number of iterations for training.
2. Specify the input parameters for the training process.
3. Execute the training command within a try-except block to handle potential errors.

Code

```
n_iterations = 2
inputs = {"topic": "CrewAI Training"}
filename = "your_model.pkl"

try:
    YourCrewName_Crew().crew().train(
      n_iterations=n_iterations,
      inputs=inputs,
      filename=filename
    )

except Exception as e:
    raise Exception(f"An error occurred while training the crew: {e}")
```

Key Points to Note

Positive Integer Requirement: Ensure that the number of iterations (`n_iterations`) is a positive integer. The code will raise a `ValueError` if this condition is not met.

Filename Requirement: Ensure that the filename ends with `.pkl` . The code will raise a `ValueError` if this condition is not met.

complexity of your agents and will also require your feedback on each iteration.

Once the training is complete, your agents will be equipped with enhanced capabilities and knowledge, ready to tackle complex tasks and provide more consistent and valuable insights.

Remember to regularly update and retrain your agents to ensure they stay up-to-date with the latest information and advancements in the field.

Happy training with CrewAI! 🚀

‹ Collaboration Memory ›

Core Concepts

Memory

Leveraging memory systems in the CrewAI framework to enhance agent capabilities.

Introduction to Memory Systems in CrewAI

The crewAI framework introduces a sophisticated memory system designed to significantly enhance the capabilities of AI agents. This system comprises `short-term memory`, `long-term memory`, `entity memory`, and `contextual memory`, each serving a unique purpose in aiding agents to remember, reason, and learn from past interactions.

Memory System Components

Component	Description
Short-Term Memory	Temporarily stores recent interactions and outcomes using `RAG`, enabling agents to recall and utilize information relevant to their current context during the current executions.
Long-Term Memory	Preserves valuable insights and learnings from past executions, allowing agents to build and refine their knowledge over time.
Entity Memory	Captures and organizes information about entities (people, places, concepts) encountered during tasks, facilitating deeper understanding and relationship mapping. Uses `RAG` for storing entity information.
Contextual Memory	Maintains the context of interactions by combining `ShortTermMemory`, `LongTermMemory`, and `EntityMemory`, aiding in the coherence and relevance of agent responses over a sequence of tasks or a conversation.
User Memory	Stores user-specific information and preferences, enhancing personalization and user experience.

relevant responses.

2. **Experience Accumulation**: Long-term memory allows agents to accumulate experiences, learning from past actions to improve future decision-making and problem-solving.

3. **Entity Understanding**: By maintaining entity memory, agents can recognize and remember key entities, enhancing their ability to process and interact with complex information.

Implementing Memory in Your Crew

When configuring a crew, you can enable and customize each memory component to suit the crew's objectives and the nature of tasks it will perform. By default, the memory system is disabled, and you can ensure it is active by setting `memory=True` in the crew configuration. The memory will use OpenAI embeddings by default, but you can change it by setting `embedder` to a different model. It's also possible to initialize the memory instance with your own instance.

The 'embedder' only applies to **Short-Term Memory** which uses Chroma for RAG. The **Long-Term Memory** uses SQLite3 to store task results. Currently, there is no way to override these storage implementations. The data storage files are saved into a platform-specific location found using the appdirs package, and the name of the project can be overridden using the **CREWAI_STORAGE_DIR** environment variable.

Example: Configuring Memory for a Crew

Code

```
from crewai import Crew, Agent, Task, Process
```

```
    memory=True,
    verbose=True
)
```

Example: Use Custom Memory Instances e.g FAISS as the VectorDB

Code

```
from crewai import Crew, Process
from crewai.memory import LongTermMemory, ShortTermMemory, EntityMemory
from crewai.memory.storage import LTMSQLiteStorage, RAGStorage
from typing import List, Optional

# Assemble your crew with memory capabilities
my_crew: Crew = Crew(
    agents = [...],
    tasks = [...],
    process = Process.sequential,
    memory = True,
    # Long-term memory for persistent storage across sessions
    long_term_memory = LongTermMemory(
        storage=LTMSQLiteStorage(
            db_path="/my_crew1/long_term_memory_storage.db"
        )
    ),
    # Short-term memory for current context using RAG
    short_term_memory = ShortTermMemory(
        storage = RAGStorage(
                embedder_config={
                    "provider": "openai",
                    "config": {
                        "model": 'text-embedding-3-small'
                    }
```

```
    ),
    # Entity memory for tracking key information about entities
    entity_memory = EntityMemory(
        storage=RAGStorage(
            embedder_config={
                "provider": "openai",
                "config": {
                    "model": 'text-embedding-3-small'
                }
            },
            type="short_term",
            path="/my_crew1/"
        )
    ),
    verbose=True,
)
```

Security Considerations

When configuring memory storage:

- Use environment variables for storage paths (e.g., `CREWAI_STORAGE_DIR`)
- Never hardcode sensitive information like database credentials
- Consider access permissions for storage directories
- Use relative paths when possible to maintain portability

Example using environment variables:

```
import os
from crewai import Crew
from crewai.memory import LongTermMemory
```

```
    memory=True,
    long_term_memory=LongTermMemory(
        storage=LTMSQLiteStorage(
            db_path="{storage_path}/memory.db".format(storage_path=storage_path)
        )
    )
)
```

Configuration Examples

Basic Memory Configuration

```
from crewai import Crew
from crewai.memory import LongTermMemory

# Simple memory configuration
crew = Crew(memory=True)  # Uses default storage locations
```

Custom Storage Configuration

```
from crewai import Crew
from crewai.memory import LongTermMemory
from crewai.memory.storage import LTMSQLiteStorage

# Configure custom storage paths
crew = Crew(
    memory=True,
    long_term_memory=LongTermMemory(
        storage=LTMSQLiteStorage(db_path="./memory.db")
```

crewai

Core Concepts › Memory

Integrating Mem0 for Enhanced User Memory

Mem0 is a self-improving memory layer for LLM applications, enabling personalized AI experiences.

To include user-specific memory you can get your API key here and refer the docs for adding user preferences.

Code

```
import os
from crewai import Crew, Process
from mem0 import MemoryClient

# Set environment variables for Mem0
os.environ["MEM0_API_KEY"] = "m0-xx"

# Step 1: Record preferences based on past conversation or user input
client = MemoryClient()
messages = [
    {"role": "user", "content": "Hi there! I'm planning a vacation and could use
    {"role": "assistant", "content": "Hello! I'd be happy to help with your vacat
    {"role": "user", "content": "I am more of a beach person than a mountain pers
    {"role": "assistant", "content": "That's interesting. Do you like hotels or A
    {"role": "user", "content": "I like Airbnb more."},
]
client.add(messages, user_id="john")

# Step 2: Create a Crew with User Memory

crew = Crew(
    agents=[...],
    tasks=[...],
    verbose=True,
    process=Process.sequential,
```

```
)
```

Memory Configuration Options

If you want to access a specific organization and project, you can set the `org_id` and `project_id` parameters in the memory configuration.

Code

```
from crewai import Crew

crew = Crew(
    agents=[...],
    tasks=[...],
    verbose=True,
    memory=True,
    memory_config={
        "provider": "mem0",
        "config": {"user_id": "john", "org_id": "my_org_id", "project_id": "my_pr
    },
)
```

Additional Embedding Providers

Using OpenAI embeddings (already default)

Code

```
from crewai import Crew, Agent, Task, Process
```

```
    verbose=True,
    embedder={
        "provider": "openai",
        "config": {
            "model": 'text-embedding-3-small'
        }
    }
)
```

Alternatively, you can directly pass the OpenAIEmbeddingFunction to the embedder parameter.

Example:

Code

```
from crewai import Crew, Agent, Task, Process
from chromadb.utils.embedding_functions import OpenAIEmbeddingFunction

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "openai",
        "config": {
            "model": 'text-embedding-3-small'
        }
    }
)
```

```
from crewai import Crew, Agent, Task, Process

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "ollama",
        "config": {
            "model": "mxbai-embed-large"
        }
    }
)
```

Using Google AI embeddings

Prerequisites

Before using Google AI embeddings, ensure you have:

- Access to the Gemini API
- The necessary API keys and permissions

You will need to update your *pyproject.toml* dependencies:

```
dependencies = [
    "google-generativeai>=0.8.4", #main version in January/2025 - crewai v.0.100.
    "crewai[tools]>=0.100.0,<1.0.0"
]
```

```
my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "google",
        "config": {
            "api_key": "<YOUR_API_KEY>",
            "model": "<model_name>"
        }
    }
)
```

Using Azure OpenAI embeddings

Code

```
from chromadb.utils.embedding_functions import OpenAIEmbeddingFunction
from crewai import Crew, Agent, Task, Process

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "openai",
        "config": {
            "api_key": "YOUR_API_KEY",
            "api_base": "YOUR_API_BASE_PATH",
```

Using Vertex AI embeddings

Code

```
from chromadb.utils.embedding_functions import GoogleVertexEmbeddingFunction
from crewai import Crew, Agent, Task, Process

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "vertexai",
        "config": {
            "project_id"="YOUR_PROJECT_ID",
            "region"="YOUR_REGION",
            "api_key"="YOUR_API_KEY",
            "model_name"="textembedding-gecko"
        }
    }
)
```

Using Cohere embeddings

Code

```
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "cohere",
        "config": {
            "api_key": "YOUR_API_KEY",
            "model": "<model_name>"
        }
    }
)
```

Using VoyageAI embeddings

Code

```
from crewai import Crew, Agent, Task, Process

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "voyageai",
        "config": {
            "api_key": "YOUR_API_KEY",
            "model": "<model_name>"
        }
```

crewai

Core Concepts › Memory

Using HuggingFace embeddings

Code

```
from crewai import Crew, Agent, Task, Process

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "huggingface",
        "config": {
            "api_url": "<api_url>",
        }
    }
)
```

Using Watson embeddings

Code

```
from crewai import Crew, Agent, Task, Process

# Note: Ensure you have installed and imported `ibm_watsonx_ai` for Watson embedd

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
```

```
            "model": "<model_name>",
            "api_url": "<api_url>",
            "api_key": "<YOUR_API_KEY>",
            "project_id": "<YOUR_PROJECT_ID>",
        }
    }
)
```

Using Amazon Bedrock embeddings

Code

```
# Note: Ensure you have installed `boto3` for Bedrock embeddings to work.

import os
import boto3
from crewai import Crew, Agent, Task, Process

boto3_session = boto3.Session(
    region_name=os.environ.get("AWS_REGION_NAME"),
    aws_access_key_id=os.environ.get("AWS_ACCESS_KEY_ID"),
    aws_secret_access_key=os.environ.get("AWS_SECRET_ACCESS_KEY")
)

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    embedder={
    "provider": "bedrock",
        "config":{
            "session": boto3_session,
```

```
)
```

Adding Custom Embedding Function

Code

```
from crewai import Crew, Agent, Task, Process
from chromadb import Documents, EmbeddingFunction, Embeddings

# Create a custom embedding function
class CustomEmbedder(EmbeddingFunction):
    def __call__(self, input: Documents) -> Embeddings:
        # generate embeddings
        return [1, 2, 3] # this is a dummy embedding

my_crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True,
    verbose=True,
    embedder={
        "provider": "custom",
        "config": {
            "embedder": CustomEmbedder()
        }
    }
)
```

Resetting Memory

Core Concepts › Memory

Option	Description	Type	Default
`-l , --long`	Reset LONG TERM memory.	Flag (boolean)	False
`-s , --short`	Reset SHORT TERM memory.	Flag (boolean)	False
`-e , --entities`	Reset ENTITIES memory.	Flag (boolean)	False
`-k , --kickoff-outputs`	Reset LATEST KICKOFF TASK OUTPUTS.	Flag (boolean)	False
`-a , --all`	Reset ALL memories.	Flag (boolean)	False

Benefits of Using CrewAI's Memory System

- 🦾 **Adaptive Learning:** Crews become more efficient over time, adapting to new information and refining their approach to tasks.
- 🤌 **Enhanced Personalization:** Memory enables agents to remember user preferences and historical interactions, leading to personalized experiences.
- 🧠 **Improved Problem Solving:** Access to a rich memory store aids agents in making more informed decisions, drawing on past learnings and contextual insights.

Conclusion

Integrating CrewAI's memory system into your projects is straightforward. By leveraging the provided memory components and configurations, you can quickly empower your agents with the ability to remember, reason, and learn from their interactions, unlocking new levels of intelligence and capability.

Was this page helpful? Yes No

Powered by Mintlify

Core Concepts

Planning

Learn how to add planning to your CrewAI Crew and improve their performance.

Introduction

The planning feature in CrewAI allows you to add planning capability to your crew. When enabled, before each Crew iteration, all Crew information is sent to an AgentPlanner that will plan the tasks step by step, and this plan will be added to each task description.

Using the Planning Feature

Getting started with the planning feature is very easy, the only step required is to add `planning=True` to your Crew:

Code

```
from crewai import Crew, Agent, Task, Process

# Assemble your crew with planning capabilities
my_crew = Crew(
    agents=self.agents,
    tasks=self.tasks,
    process=Process.sequential,
    planning=True,
)
```

From this point on, your crew will have planning enabled, and the tasks will be planned before each iteration.

represents the output of the `AgentPlanner` responsible for creating the step-by-step logic to add to the Agents' tasks.

Code Result

```
from crewai import Crew, Agent, Task, Process

# Assemble your crew with planning capabilities and custom LLM
my_crew = Crew(
    agents=self.agents,
    tasks=self.tasks,
    process=Process.sequential,
    planning=True,
    planning_llm="gpt-4o"
)

# Run the crew
my_crew.kickoff()
```

Was this page helpful? Yes No

‹ Memory Testing ›

Core Concepts › Planning

Core Concepts

Testing

Learn how to test your CrewAI Crew and evaluate their performance.

Introduction

Testing is a crucial part of the development process, and it is essential to ensure that your crew is performing as expected. With crewAI, you can easily test your crew and evaluate its performance using the built-in testing capabilities.

Using the Testing Feature

We added the CLI command `crewai test` to make it easy to test your crew. This command will run your crew for a specified number of iterations and provide detailed performance metrics. The parameters are `n_iterations` and `model`, which are optional and default to 2 and `gpt-4o-mini` respectively. For now, the only provider available is OpenAI.

```
crewai test
```

If you want to run more iterations or use a different model, you can specify the parameters like this:

```
crewai test --n_iterations 5 --model gpt-4o
```

or using the short forms:

number of iterations, and the performance metrics will be displayed at the end of the run.

A table of scores at the end will show the performance of the crew in terms of the following metrics:

Tasks/Crew/Agents	Run 1	Run 2	Avg. Total	Agents	Additional Info
Task 1	9.0	9.5	9.2	Professional Insights	
				Researcher	
Task 2	9.0	10.0	9.5	Company Profile Investigator	
Task 3	9.0	9.0	9.0	Automation Insights	
				Specialist	
Task 4	9.0	9.0	9.0	Final Report Compiler	Automation Insights Specialist
Crew	9.00	9.38	9.2		
Execution Time (s)	126	145	135		

The example above shows the test results for two runs of the crew with two tasks, with the average total score for each task and the crew as a whole.

‹ Planning CLI ›

Core Concepts

CLI

Learn how to use the CrewAI CLI to interact with CrewAI.

CrewAI CLI Documentation

The CrewAI CLI provides a set of commands to interact with CrewAI, allowing you to create, train, run, and manage crews & flows.

Installation

To use the CrewAI CLI, make sure you have CrewAI installed:

Terminal

```
pip install crewai
```

Basic Usage

The basic structure of a CrewAI CLI command is:

Terminal

```
crewai [COMMAND] [OPTIONS] [ARGUMENTS]
```

Core Concepts › CLI

Terminal

```
crewai create [OPTIONS] TYPE NAME
```

`TYPE` : Choose between “crew” or “flow”

`NAME` : Name of the crew or flow

Example:

Terminal

```
crewai create crew my_new_crew
crewai create flow my_new_flow
```

2. Version

Show the installed version of CrewAI.

Terminal

```
crewai version [OPTIONS]
```

`--tools` : (Optional) Show the installed version of CrewAI tools

Example:

Terminal

3. Train

Train the crew for a specified number of iterations.

Terminal

```
crewai train [OPTIONS]
```

`-n, --n_iterations INTEGER` : Number of iterations to train the crew (default: 5)

`-f, --filename TEXT` : Path to a custom file for training (default: "trained_agents_data.pkl")

Example:

Terminal

```
crewai train -n 10 -f my_training_data.pkl
```

4. Replay

Replay the crew execution from a specific task.

Terminal

```
crewai replay [OPTIONS]
```

`-t, --task_id TEXT` : Replay the crew from this task ID, including all subsequent tasks

```
crewai replay -t task_123456
```

5. Log-tasks-outputs

Retrieve your latest crew.kickoff() task outputs.

Terminal

```
crewai log-tasks-outputs
```

6. Reset-memories

Reset the crew memories (long, short, entity, latest_crew_kickoff_outputs).

Terminal

```
crewai reset-memories [OPTIONS]
```

- `-l, --long` : Reset LONG TERM memory
- `-s, --short` : Reset SHORT TERM memory
- `-e, --entities` : Reset ENTITIES memory
- `-k, --kickoff-outputs` : Reset LATEST KICKOFF TASK OUTPUTS
- `-a, --all` : Reset ALL memories

Example:

Terminal

7. Test

Test the crew and evaluate the results.

Terminal

```
crewai test [OPTIONS]
```

`-n, --n_iterations INTEGER` : Number of iterations to test the crew (default: 3)

`-m, --model TEXT` : LLM Model to run the tests on the Crew (default: “gpt-4o-mini”)

Example:

Terminal

```
crewai test -n 5 -m gpt-3.5-turbo
```

8. Run

Run the crew.

Terminal

```
crewai run
```

Make sure to run these commands from the directory where your CrewAI project is set up. Some commands may require additional configuration or setup within your project structure.

After receiving the results, you can continue interacting with the assistant for further instructions or questions.

Terminal

```
crewai chat
```

Ensure you execute these commands from your CrewAI project's root directory.

IMPORTANT: Set the `chat_llm` property in your `crew.py` file to enable this command.

```
@crew
def crew(self) -> Crew:
    return Crew(
        agents=self.agents,
        tasks=self.tasks,
        process=Process.sequential,
        verbose=True,
        chat_llm="gpt-4o",  # LLM for chat orchestration
    )
```

10. API Keys

When running `crewai create crew` command, the CLI will first show you the top 5 most common LLM providers and ask you to select one.

OpenAI

Groq

Anthropic

Google Gemini

SambaNova

When you select a provider, the CLI will prompt you to enter your API key.

Other Options

If you select option 6, you will be able to select from a list of LiteLLM supported providers.

When you select a provider, the CLI will prompt you to enter the Key name and the API key.

See the following link for each provider's key name:

LiteLLM Providers

Core Concepts › CLI

Core Concepts

Tools

Understanding and leveraging tools within the CrewAI framework for agent collaboration and task execution.

Introduction

CrewAI tools empower agents with capabilities ranging from web searching and data analysis to collaboration and delegating tasks among coworkers. This documentation outlines how to create, integrate, and leverage these tools within the CrewAI framework, including a new focus on collaboration tools.

What is a Tool?

A tool in CrewAI is a skill or function that agents can utilize to perform various actions. This includes tools from the CrewAI Toolkit and LangChain Tools, enabling everything from simple searches to complex interactions and effective teamwork among agents.

Key Characteristics of Tools

- **Utility**: Crafted for tasks such as web searching, data analysis, content generation, and agent collaboration.
- **Integration**: Boosts agent capabilities by seamlessly integrating tools into their workflow.
- **Customizability**: Provides the flexibility to develop custom tools or utilize existing ones, catering to the specific needs of agents.
- **Error Handling**: Incorporates robust error handling mechanisms to ensure smooth operation.

Core Concepts › Tools

Using CrewAI Tools

To enhance your agents' capabilities with crewAI tools, begin by installing our extra tools package:

```
pip install 'crewai[tools]'
```

Here's an example demonstrating their use:

Code

```
import os
from crewai import Agent, Task, Crew
# Importing crewAI tools
from crewai_tools import (
    DirectoryReadTool,
    FileReadTool,
    SerperDevTool,
    WebsiteSearchTool
)

# Set up API keys
os.environ["SERPER_API_KEY"] = "Your Key" # serper.dev API key
os.environ["OPENAI_API_KEY"] = "Your Key"

# Instantiate tools
docs_tool = DirectoryReadTool(directory='./blog-posts')
file_tool = FileReadTool()
search_tool = SerperDevTool()
web_rag_tool = WebsiteSearchTool()

# Create agents
researcher = Agent(
```

```
)

writer = Agent(
    role='Content Writer',
    goal='Craft engaging blog posts about the AI industry',
    backstory='A skilled writer with a passion for technology.',
    tools=[docs_tool, file_tool],
    verbose=True
)

# Define tasks
research = Task(
    description='Research the latest trends in the AI industry and provide a summ
    expected_output='A summary of the top 3 trending developments in the AI indus
    agent=researcher
)

write = Task(
    description='Write an engaging blog post about the AI industry, based on the
    expected_output='A 4-paragraph blog post formatted in markdown with engaging,
    agent=writer,
    output_file='blog-posts/new_post.md'  # The final blog post will be saved her
)

# Assemble a crew with planning enabled
crew = Crew(
    agents=[researcher, writer],
    tasks=[research, write],
    verbose=True,
    planning=True,  # Enable planning feature
)

# Execute tasks
crew.kickoff()
```

Caching Mechanism: All tools support caching, enabling agents to efficiently reuse previously obtained results, reducing the load on external resources and speeding up the execution time. You can also define finer control over the caching mechanism using the `cache_function` attribute on the tool.

Here is a list of the available tools and their descriptions:

Tool	Description
BrowserbaseLoadTool	A tool for interacting with and extracting data from web browsers.
CodeDocsSearchTool	A RAG tool optimized for searching through code documentation and related technical documents.
CodeInterpreterTool	A tool for interpreting python code.
ComposioTool	Enables use of Composio tools.
CSVSearchTool	A RAG tool designed for searching within CSV files, tailored to handle structured data.
DALL-E Tool	A tool for generating images using the DALL-E API.
DirectorySearchTool	A RAG tool for searching within directories, useful for navigating through file systems.
DOCXSearchTool	A RAG tool aimed at searching within DOCX documents, ideal for processing Word files.
DirectoryReadTool	Facilitates reading and processing of directory structures and their contents.
EXASearchTool	A tool designed for performing exhaustive searches across various data sources.
FileReadTool	Enables reading and extracting data from files, supporting various file formats.
FirecrawlSearchTool	A tool to search webpages using Firecrawl and return the results.
FirecrawlCrawlWebsiteTool	A tool for crawling webpages using Firecrawl.

	and documentation search.
SerperDevTool	A specialized tool for development purposes, with specific functionalities under development.
TXTSearchTool	A RAG tool focused on searching within text (.txt) files, suitable for unstructured data.
JSONSearchTool	A RAG tool designed for searching within JSON files, catering to structured data handling.
LlamaIndexTool	Enables the use of LlamaIndex tools.
MDXSearchTool	A RAG tool tailored for searching within Markdown (MDX) files, useful for documentation.
PDFSearchTool	A RAG tool aimed at searching within PDF documents, ideal for processing scanned documents.
PGSearchTool	A RAG tool optimized for searching within PostgreSQL databases, suitable for database queries.
Vision Tool	A tool for generating images using the DALL-E API.
RagTool	A general-purpose RAG tool capable of handling various data sources and types.
ScrapeElementFromWebsiteTool	Enables scraping specific elements from websites, useful for targeted data extraction.
ScrapeWebsiteTool	Facilitates scraping entire websites, ideal for comprehensive data collection.
WebsiteSearchTool	A RAG tool for searching website content, optimized for web data extraction.
XMLSearchTool	A RAG tool designed for searching within XML files, suitable for structured data formats.
YoutubeChannelSearchTool	A RAG tool for searching within YouTube channels, useful for video content analysis.
YoutubeVideoSearchTool	A RAG tool aimed at searching within YouTube videos, ideal for video data extraction.

crewai

Core Concepts › Tools

There are two main ways for one to create a CrewAI tool:

Subclassing `BaseTool`

Code

```
from crewai.tools import BaseTool
from pydantic import BaseModel, Field

class MyToolInput(BaseModel):
    """Input schema for MyCustomTool."""
    argument: str = Field(..., description="Description of the argument.")

class MyCustomTool(BaseTool):
    name: str = "Name of my tool"
    description: str = "What this tool does. It's vital for effective utilization
    args_schema: Type[BaseModel] = MyToolInput

    def _run(self, argument: str) -> str:
        # Your tool's logic here
        return "Tool's result"
```

Utilizing the `tool` Decorator

Code

```
from crewai.tools import tool
@tool("Name of my tool")
def my_tool(question: str) -> str:
    """Clear description for what this tool is useful for, your agent will need t
```

Structured Tools

The `StructuredTool` class wraps functions as tools, providing flexibility and validation while reducing boilerplate. It supports custom schemas and dynamic logic for seamless integration of complex functionalities.

Example:

Using `StructuredTool.from_function`, you can wrap a function that interacts with an external API or system, providing a structured interface. This enables robust validation and consistent execution, making it easier to integrate complex functionalities into your applications as demonstrated in the following example:

```
from crewai.tools.structured_tool import CrewStructuredTool
from pydantic import BaseModel

# Define the schema for the tool's input using Pydantic
class APICallInput(BaseModel):
    endpoint: str
    parameters: dict

# Wrapper function to execute the API call
def tool_wrapper(*args, **kwargs):
    # Here, you would typically call the API using the parameters
    # For demonstration, we'll return a placeholder string
    return f"Call the API at {kwargs['endpoint']} with parameters {kwargs['parame

# Create and return the structured tool
def create_structured_tool():
    return CrewStructuredTool.from_function(
        name='Wrapper API',
        description="A tool to wrap API calls with structured input.",
        args_schema=APICallInput,
        func=tool_wrapper,
```

```
# Execute the tool with structured input
result = structured_tool._run(**{
    "endpoint": "https://example.com/api",
    "parameters": {"key1": "value1", "key2": "value2"}
})
print(result)  # Output: Call the API at https://example.com/api with parameters
```

Custom Caching Mechanism

Tools can optionally implement a `cache_function` to fine-tune caching behavior. This function determines when to cache results based on specific conditions, offering granular control over caching logic.

Code

```
from crewai.tools import tool

@tool
def multiplication_tool(first_number: int, second_number: int) -> str:
    """Useful for when you need to multiply two numbers together."""
    return first_number * second_number

def cache_func(args, result):
    # In this case, we only cache the result if it's a multiple of 2
    cache = result % 2 == 0
    return cache

multiplication_tool.cache_function = cache_func

writer1 = Agent(
        role="Writer",
        goal="You write lessons of math for kids.",
```

Conclusion

Tools are pivotal in extending the capabilities of CrewAI agents, enabling them to undertake a broad spectrum of tasks and collaborate effectively. When building solutions with CrewAI, leverage both custom and existing tools to empower your agents and enhance the AI ecosystem. Consider utilizing error handling, caching mechanisms, and the flexibility of tool arguments to optimize your agents' performance and capabilities.

‹ CLI Using LangChain Tools ›

crewai

Core Concepts › Using LangChain Tools

Core Concepts

Using LangChain Tools

Learn how to integrate LangChain tools with CrewAI agents to enhance search-based queries and more.

Using LangChain Tools

CrewAI seamlessly integrates with LangChain's comprehensive list of tools, all of which can be used with CrewAI.

Code

```
import os
from dotenv import load_dotenv
from crewai import Agent, Task, Crew
from crewai.tools import BaseTool
from pydantic import Field
from langchain_community.utilities import GoogleSerperAPIWrapper

# Set up your SERPER_API_KEY key in an .env file, eg:
# SERPER_API_KEY=<your api key>
load_dotenv()

search = GoogleSerperAPIWrapper()

class SearchTool(BaseTool):
    name: str = "Search"
    description: str = "Useful for search-based queries. Use this to find current
    search: GoogleSerperAPIWrapper = Field(default_factory=GoogleSerperAPIWrapper
```

```
            return f"Error performing search: {str(e)}"

# Create Agents
researcher = Agent(
    role='Research Analyst',
    goal='Gather current market data and trends',
    backstory="""You are an expert research analyst with years of experience in
    gathering market intelligence. You're known for your ability to find
    relevant and up-to-date market information and present it in a clear,
    actionable format.""",
    tools=[SearchTool()],
    verbose=True
)

# rest of the code ...
```

Conclusion

Tools are pivotal in extending the capabilities of CrewAI agents, enabling them to undertake a broad spectrum of tasks and collaborate effectively. When building solutions with CrewAI, leverage both custom and existing tools to empower your agents and enhance the AI ecosystem. Consider utilizing error handling, caching mechanisms, and the flexibility of tool arguments to optimize your agents' performance and capabilities.

‹ Tools Using LlamaIndex Tools ›

Core Concepts

Using LlamaIndex Tools

Learn how to integrate LlamaIndex tools with CrewAI agents to enhance search-based queries and more.

Using LlamaIndex Tools

CrewAI seamlessly integrates with LlamaIndex's comprehensive toolkit for RAG (Retrieval-Augmented Generation) and agentic pipelines, enabling advanced search-based queries and more.

Here are the available built-in tools offered by LlamaIndex.

Code

```
from crewai import Agent
from crewai_tools import LlamaIndexTool

# Example 1: Initialize from FunctionTool
from llama_index.core.tools import FunctionTool

your_python_function = lambda ...: ...
og_tool = FunctionTool.from_defaults(
    your_python_function,
    name="<name>",
    description='<description>'
)
tool = LlamaIndexTool.from_tool(og_tool)

# Example 2: Initialize from LlamaHub Tools
from llama_index.tools.wolfram_alpha import WolframAlphaToolSpec
```

```
query_engine = index.as_query_engine()
query_tool = LlamaIndexTool.from_query_engine(
    query_engine,
    name="Uber 2019 10K Query Tool",
    description="Use this tool to lookup the 2019 Uber 10K Annual Report"
)

# Create and assign the tools to an agent
agent = Agent(
    role='Research Analyst',
    goal='Provide up-to-date market analysis',
    backstory='An expert analyst with a keen eye for market trends.',
    tools=[tool, *tools, query_tool]
)

# rest of the code ...
```

Steps to Get Started

To effectively use the LlamaIndexTool, follow these steps:

1 Package Installation

Make sure that `crewai[tools]` package is installed in your Python environment:

Terminal

```
pip install 'crewai[tools]'
```

2 Install and Use LlamaIndex

Was this page helpful? Yes No

‹ Using LangChain Tools

Create Custom Tools ›

How to Guides

Create Custom Tools

Comprehensive guide on crafting, using, and managing custom tools within the CrewAI framework, including new functionalities and error handling.

Creating and Utilizing Tools in CrewAI

This guide provides detailed instructions on creating custom tools for the CrewAI framework and how to efficiently manage and utilize these tools, incorporating the latest functionalities such as tool delegation, error handling, and dynamic tool calling. It also highlights the importance of collaboration tools, enabling agents to perform a wide range of actions.

Subclassing `BaseTool`

To create a personalized tool, inherit from `BaseTool` and define the necessary attributes, including the `args_schema` for input validation, and the `_run` method.

Code

```
from typing import Type
from crewai.tools import BaseTool
from pydantic import BaseModel, Field

class MyToolInput(BaseModel):
    """Input schema for MyCustomTool."""
    argument: str = Field(..., description="Description of the argument.")

class MyCustomTool(BaseTool):
    name: str = "Name of my tool"
    description: str = "What this tool does. It's vital for effective utilization
```

Using the `tool` Decorator

Alternatively, you can use the tool decorator `@tool`. This approach allows you to define the tool's attributes and functionality directly within a function, offering a concise and efficient way to create specialized tools tailored to your needs.

Code

```
from crewai.tools import tool

@tool("Tool Name")
def my_simple_tool(question: str) -> str:
    """Tool description for clarity."""
    # Tool logic here
    return "Tool output"
```

Defining a Cache Function for the Tool

To optimize tool performance with caching, define custom caching strategies using the `cache_function` attribute.

Code

```
@tool("Tool with Caching")
def cached_tool(argument: str) -> str:
    """Tool functionality description."""
    return "Cacheable result"

def my_cache_strategy(arguments: dict, result: str) -> bool:
```

By adhering to these guidelines and incorporating new functionalities and collaboration tools into your tool creation and management processes, you can leverage the full capabilities of the CrewAI framework, enhancing both the development experience and the efficiency of your AI agents.

‹ Using LlamaIndex Tools Sequential Processes ›

How to Guides

Sequential Processes

A comprehensive guide to utilizing the sequential processes for task execution in CrewAI projects.

Introduction

CrewAI offers a flexible framework for executing tasks in a structured manner, supporting both sequential and hierarchical processes. This guide outlines how to effectively implement these processes to ensure efficient task execution and project completion.

Sequential Process Overview

The sequential process ensures tasks are executed one after the other, following a linear progression. This approach is ideal for projects requiring tasks to be completed in a specific order.

Key Features

- **Linear Task Flow**: Ensures orderly progression by handling tasks in a predetermined sequence.
- **Simplicity**: Best suited for projects with clear, step-by-step tasks.
- **Easy Monitoring**: Facilitates easy tracking of task completion and project progress.

Implementing the Sequential Process

crewai

How to Guides › Sequential Processes

```python
from crewai import Crew, Process, Agent, Task, TaskOutput, CrewOutput

# Define your agents
researcher = Agent(
  role='Researcher',
  goal='Conduct foundational research',
  backstory='An experienced researcher with a passion for uncovering insights'
)
analyst = Agent(
  role='Data Analyst',
  goal='Analyze research findings',
  backstory='A meticulous analyst with a knack for uncovering patterns'
)
writer = Agent(
  role='Writer',
  goal='Draft the final report',
  backstory='A skilled writer with a talent for crafting compelling narratives'
)

# Define your tasks
research_task = Task(
  description='Gather relevant data...',
  agent=researcher,
  expected_output='Raw Data'
)
analysis_task = Task(
  description='Analyze the data...',
  agent=analyst,
  expected_output='Data Insights'
)
writing_task = Task(
  description='Compose the report...',
  agent=writer,
  expected_output='Final Report'
)
```

```
  process=Process.sequential
)

# Execute the crew
result = report_crew.kickoff()

# Accessing the type-safe output
task_output: TaskOutput = result.tasks[0].output
crew_output: CrewOutput = result.output
```

Note:

Each task in a sequential process **must** have an agent assigned. Ensure that every `Task` includes an `agent` parameter.

Workflow in Action

1. **Initial Task**: In a sequential process, the first agent completes their task and signals completion.
2. **Subsequent Tasks**: Agents pick up their tasks based on the process type, with outcomes of preceding tasks or directives guiding their execution.
3. **Completion**: The process concludes once the final task is executed, leading to project completion.

Advanced Features

Task Delegation

In sequential processes, if an agent has `allow_delegation` set to `True`, they can delegate tasks to other agents in the crew. This feature is automatically set up when there are multiple

Tasks can be executed asynchronously, allowing for parallel processing when appropriate. To create an asynchronous task, set `async_execution=True` when defining the task.

Memory and Caching

CrewAI supports both memory and caching features:

- **Memory**: Enable by setting `memory=True` when creating the Crew. This allows agents to retain information across tasks.
- **Caching**: By default, caching is enabled. Set `cache=False` to disable it.

Callbacks

You can set callbacks at both the task and step level:

- `task_callback` : Executed after each task completion.
- `step_callback` : Executed after each step in an agent's execution.

Usage Metrics

CrewAI tracks token usage across all tasks and agents. You can access these metrics after execution.

Best Practices for Sequential Processes

1. **Order Matters**: Arrange tasks in a logical sequence where each task builds upon the previous one.
2. **Clear Task Descriptions**: Provide detailed descriptions for each task to guide the agents effectively.

This updated documentation ensures that details accurately reflect the latest changes in the codebase and clearly describes how to leverage new features and configurations. The content is kept simple and direct to ensure easy understanding.

Was this page helpful? Yes No

‹ Create Custom Tools Hierarchical Process ›

How to Guides

Hierarchical Process

A comprehensive guide to understanding and applying the hierarchical process within your CrewAI projects, updated to reflect the latest coding practices and functionalities.

Introduction

The hierarchical process in CrewAI introduces a structured approach to task management, simulating traditional organizational hierarchies for efficient task delegation and execution. This systematic workflow enhances project outcomes by ensuring tasks are handled with optimal efficiency and accuracy.

> The hierarchical process is designed to leverage advanced models like GPT-4, optimizing token usage while handling complex tasks with greater efficiency.

Hierarchical Process Overview

By default, tasks in CrewAI are managed through a sequential process. However, adopting a hierarchical approach allows for a clear hierarchy in task management, where a 'manager' agent coordinates the workflow, delegates tasks, and validates outcomes for streamlined and effective execution. This manager agent can now be either automatically created by CrewAI or explicitly set by the user.

Key Features

Efficient Workflow: Emulates corporate structures, providing an organized approach to task management.

System Prompt Handling: Optionally specify whether the system should use predefined prompts.

Stop Words Control: Optionally specify whether stop words should be used, supporting various models including the o1 models.

Context Window Respect: Prioritize important context by enabling respect of the context window, which is now the default behavior.

Delegation Control: Delegation is now disabled by default to give users explicit control.

Max Requests Per Minute: Configurable option to set the maximum number of requests per minute.

Max Iterations: Limit the maximum number of iterations for obtaining a final answer.

Implementing the Hierarchical Process

To utilize the hierarchical process, it's essential to explicitly set the process attribute to `Process.hierarchical`, as the default behavior is `Process.sequential`. Define a crew with a designated manager and establish a clear chain of command.

Assign tools at the agent level to facilitate task delegation and execution by the designated agents under the manager's guidance. Tools can also be specified at the task level for precise control over tool availability during task execution.

Configuring the `manager_llm` parameter is crucial for the hierarchical process. The system requires a manager LLM to be set up for proper function, ensuring tailored decision-making.

Code

How to Guides › Hierarchical Process

```
researcher = Agent(
    role='Researcher',
    goal='Conduct in-depth analysis',
    backstory='Experienced data analyst with a knack for uncovering hidden trends
    cache=True,
    verbose=False,
    # tools=[]  # This can be optionally specified; defaults to an empty list
    use_system_prompt=True,  # Enable or disable system prompts for this agent
    max_rpm=30,  # Limit on the number of requests per minute
    max_iter=5  # Maximum number of iterations for a final answer
)
writer = Agent(
    role='Writer',
    goal='Create engaging content',
    backstory='Creative writer passionate about storytelling in technical domains
    cache=True,
    verbose=False,
    # tools=[]  # Optionally specify tools; defaults to an empty list
    use_system_prompt=True,  # Enable or disable system prompts for this agent
    max_rpm=30,  # Limit on the number of requests per minute
    max_iter=5  # Maximum number of iterations for a final answer
)

# Establishing the crew with a hierarchical process and additional configurations
project_crew = Crew(
    tasks=[...],  # Tasks to be delegated and executed under the manager's superv
    agents=[researcher, writer],
    manager_llm=ChatOpenAI(temperature=0, model="gpt-4"),  # Mandatory if manager
    process=Process.hierarchical,  # Specifies the hierarchical management approa
    respect_context_window=True,  # Enable respect of the context window for task
    memory=True,  # Enable memory usage for enhanced task execution
    manager_agent=None,  # Optional: explicitly set a specific agent as manager i
    planning=True,  # Enable planning feature for pre-execution strategy
)
```

2. **Execution and Review**: Agents complete their tasks with the option for asynchronous execution and callback functions for streamlined workflows.
3. **Sequential Task Progression**: Despite being a hierarchical process, tasks follow a logical order for smooth progression, facilitated by the manager's oversight.

Conclusion

Adopting the hierarchical process in CrewAI, with the correct configurations and understanding of the system's capabilities, facilitates an organized and efficient approach to project management. Utilize the advanced features and customizations to tailor the workflow to your specific needs, ensuring optimal task execution and project success.

‹ Sequential Processes Create Your Own Manager Agent ›

How to Guides

Create Your Own Manager Agent

Learn how to set a custom agent as the manager in CrewAI, providing more control over task management and coordination.

Setting a Specific Agent as Manager in CrewAI

CrewAI allows users to set a specific agent as the manager of the crew, providing more control over the management and coordination of tasks. This feature enables the customization of the managerial role to better fit your project's requirements.

Using the `manager_agent` Attribute

Custom Manager Agent

The `manager_agent` attribute allows you to define a custom agent to manage the crew. This agent will oversee the entire process, ensuring that tasks are completed efficiently and to the highest standard.

Example

Code

```
import os
from crewai import Agent, Task, Crew, Process

# Define your agents
researcher = Agent(
```

```
writer = Agent(
    role="Senior Writer",
    goal="Create compelling content about AI and AI agents",
    backstory="You're a senior writer, specialized in technology, software engine
    allow_delegation=False,
)

# Define your task
task = Task(
    description="Generate a list of 5 interesting ideas for an article, then writ
    expected_output="5 bullet points, each with a paragraph and accompanying note
)

# Define the manager agent
manager = Agent(
    role="Project Manager",
    goal="Efficiently manage the crew and ensure high-quality task completion",
    backstory="You're an experienced project manager, skilled in overseeing compl
    allow_delegation=True,
)

# Instantiate your crew with a custom manager
crew = Crew(
    agents=[researcher, writer],
    tasks=[task],
    manager_agent=manager,
    process=Process.hierarchical,
)

# Start the crew's work
result = crew.kickoff()
```

Improved Coordination: Ensure efficient task coordination and management by an experienced agent.

Customizable Management: Define managerial roles and responsibilities that align with your project's goals.

Setting a Manager LLM

If you're using the hierarchical process and don't want to set a custom manager agent, you can specify the language model for the manager:

Code

```
from crewai import LLM

manager_llm = LLM(model="gpt-4o")

crew = Crew(
    agents=[researcher, writer],
    tasks=[task],
    process=Process.hierarchical,
    manager_llm=manager_llm
)
```

Either `manager_agent` or `manager_llm` must be set when using the hierarchical process.

Was this page helpful? Yes No

How to Guides

Connect to any LLM

Comprehensive guide on integrating CrewAI with various Large Language Models (LLMs) using LiteLLM, including supported providers and configuration options.

Connect CrewAI to LLMs

CrewAI uses LiteLLM to connect to a wide variety of Language Models (LLMs). This integration provides extensive versatility, allowing you to use models from numerous providers with a simple, unified interface.

By default, CrewAI uses the `gpt-4o-mini` model. This is determined by the `OPENAI_MODEL_NAME` environment variable, which defaults to "gpt-4o-mini" if not set. You can easily configure your agents to use a different model or provider as described in this guide.

Supported Providers

LiteLLM supports a wide range of providers, including but not limited to:

- OpenAI
- Anthropic
- Google (Vertex AI, Gemini)
- Azure OpenAI
- AWS (Bedrock, SageMaker)
- Cohere
- VoyageAI

Replicate

Together AI

AI21

Cloudflare Workers AI

DeepInfra

Groq

SambaNova

NVIDIA NIMs

And many more!

For a complete and up-to-date list of supported providers, please refer to the LiteLLM Providers documentation.

Changing the LLM

To use a different LLM with your CrewAI agents, you have several options:

Using a String Identifier | Using the LLM Class

Pass the model name as a string when initializing the agent:

Code

```
from crewai import Agent

# Using OpenAI's GPT-4
openai_agent = Agent(
    role='OpenAI Expert',
    goal='Provide insights using GPT-4',
```

```
claude_agent = Agent(
    role='Anthropic Expert',
    goal='Analyze data using Claude',
    backstory="An AI assistant leveraging Anthropic's language model.",
    llm='claude-2'
)
```

Configuration Options

When configuring an LLM for your agent, you have access to a wide range of parameters:

Parameter	Type	Description
model	`str`	The name of the model to use (e.g., “gpt-4”, “claude-2”)
temperature	`float`	Controls randomness in output (0.0 to 1.0)
max_tokens	`int`	Maximum number of tokens to generate
top_p	`float`	Controls diversity of output (0.0 to 1.0)
frequency_penalty	`float`	Penalizes new tokens based on their frequency in the text so far
presence_penalty	`float`	Penalizes new tokens based on their presence in the text so far
stop	`str , List[str]`	Sequence(s) to stop generation
base_url	`str`	The base URL for the API endpoint
api_key	`str`	Your API key for authentication

For a complete list of parameters and their descriptions, refer to the LLM class documentation.

Using Environment Variables | Using LLM Class Attributes

Code

```
import os

os.environ["OPENAI_API_KEY"] = "your-api-key"
os.environ["OPENAI_API_BASE"] = "https://api.your-provider.com/v1"
os.environ["OPENAI_MODEL_NAME"] = "your-model-name"
```

Using Local Models with Ollama

For local models like those provided by Ollama:

1 **Download and install Ollama**

Click here to download and install Ollama

2 **Pull the desired model**

For example, run `ollama pull llama3.2` to download the model.

3 **Configure your agent**

Code

```
        llm=LLM(model="ollama/llama3.2", base_url="http://localhost:11434"
    )
```

Changing the Base API URL

You can change the base API URL for any LLM provider by setting the `base_url` parameter:

Code

```
llm = LLM(
    model="custom-model-name",
    base_url="https://api.your-provider.com/v1",
    api_key="your-api-key"
)
agent = Agent(llm=llm, ...)
```

This is particularly useful when working with OpenAI-compatible APIs or when you need to specify a different endpoint for your chosen provider.

Conclusion

By leveraging LiteLLM, CrewAI offers seamless integration with a vast array of LLMs. This flexibility allows you to choose the most suitable model for your specific needs, whether you prioritize performance, cost-efficiency, or local deployment. Remember to consult the LiteLLM documentation for the most up-to-date information on supported models and configuration options.

How to Guides

Customize Agents

A comprehensive guide to tailoring agents for specific roles, tasks, and advanced customizations within the CrewAI framework.

Customizable Attributes

Crafting an efficient CrewAI team hinges on the ability to dynamically tailor your AI agents to meet the unique requirements of any project. This section covers the foundational attributes you can customize.

Key Attributes for Customization

Attribute	Description
Role	Specifies the agent's job within the crew, such as 'Analyst' or 'Customer Service Rep'.
Goal	Defines the agent's objectives, aligned with its role and the crew's overarching mission.
Backstory	Provides depth to the agent's persona, enhancing motivations and engagements within the crew.
Tools *(Optional)*	Represents the capabilities or methods the agent uses for tasks, from simple functions to complex integrations.
Cache *(Optional)*	Determines if the agent should use a cache for tool usage.
Max RPM	Sets the maximum requests per minute (`max_rpm`). Can be set to `None` for unlimited requests to external services.
Verbose *(Optional)*	Enables detailed logging for debugging and optimization, providing insights into execution processes.

	infinite loops, with a default of 25.
Max Execution Time *(Optional)*	Sets the maximum time allowed for an agent to complete a task.
System Template *(Optional)*	Defines the system format for the agent.
Prompt Template *(Optional)*	Defines the prompt format for the agent.
Response Template *(Optional)*	Defines the response format for the agent.
Use System Prompt *(Optional)*	Controls whether the agent will use a system prompt during task execution.
Respect Context Window	Enables a sliding context window by default, maintaining context size.
Max Retry Limit	Sets the maximum number of retries (`max_retry_limit`) for an agent in case of errors.

Advanced Customization Options

Beyond the basic attributes, CrewAI allows for deeper customization to enhance an agent's behavior and capabilities significantly.

Language Model Customization

Agents can be customized with specific language models (`llm`) and function-calling language models (`function_calling_llm`), offering advanced control over their processing and decision-making abilities. It's important to note that setting the `function_calling_llm` allows for overriding the default crew function-calling language model, providing a greater degree of customization.

Verbose Mode and RPM Limit

Verbose Mode: Enables detailed logging of an agent's actions, useful for debugging and optimization. Specifically, it provides insights into agent execution processes, aiding in the optimization of performance.

RPM Limit: Sets the maximum number of requests per minute (`max_rpm`). This attribute is optional and can be set to `None` for no limit, allowing for unlimited queries to external services if needed.

Maximum Iterations for Task Execution

The `max_iter` attribute allows users to define the maximum number of iterations an agent can perform for a single task, preventing infinite loops or excessively long executions. The default value is set to 25, providing a balance between thoroughness and efficiency. Once the agent approaches this number, it will try its best to give a good answer.

Customizing Agents and Tools

Agents are customized by defining their attributes and tools during initialization. Tools are critical for an agent's functionality, enabling them to perform specialized tasks. The `tools` attribute should be an array of tools the agent can utilize, and it's initialized as an empty list by default. Tools can be added or modified post-agent initialization to adapt to new requirements.

```
pip install 'crewai[tools]'
```

Example: Assigning Tools to an Agent

```
from crewai_tools import SerperDevTool

# Set API keys for tool initialization
os.environ["OPENAI_API_KEY"] = "Your Key"
os.environ["SERPER_API_KEY"] = "Your Key"

# Initialize a search tool
search_tool = SerperDevTool()

# Initialize the agent with advanced options
agent = Agent(
  role='Research Analyst',
  goal='Provide up-to-date market analysis',
  backstory='An expert analyst with a keen eye for market trends.',
  tools=[search_tool],
  memory=True, # Enable memory
  verbose=True,
  max_rpm=None, # No limit on requests per minute
  max_iter=25, # Default value for maximum iterations
)
```

Delegation and Autonomy

Controlling an agent's ability to delegate tasks or ask questions is vital for tailoring its autonomy and collaborative dynamics within the CrewAI framework. By default, the `allow_delegation` attribute is now set to `False`, disabling agents to seek assistance or delegate tasks as needed. This default behavior can be changed to promote collaborative problem-solving and efficiency within the CrewAI ecosystem. If needed, delegation can be enabled to suit specific operational requirements.

Example: Disabling Delegation for an Agent

```
    goal='Write engaging content on market trends',
    backstory='A seasoned writer with expertise in market analysis.',
    allow_delegation=True # Enabling delegation
)
```

Conclusion

Customizing agents in CrewAI by setting their roles, goals, backstories, and tools, alongside advanced options like language model customization, memory, performance settings, and delegation preferences, equips a nuanced and capable AI team ready for complex challenges.

‹ Connect to any LLM

Using Multimodal Agents ›

How to Guides

Using Multimodal Agents

Learn how to enable and use multimodal capabilities in your agents for processing images and other non-text content within the CrewAI framework.

Using Multimodal Agents

CrewAI supports multimodal agents that can process both text and non-text content like images. This guide will show you how to enable and use multimodal capabilities in your agents.

Enabling Multimodal Capabilities

To create a multimodal agent, simply set the `multimodal` parameter to `True` when initializing your agent:

```
from crewai import Agent

agent = Agent(
    role="Image Analyst",
    goal="Analyze and extract insights from images",
    backstory="An expert in visual content interpretation with years of experienc
    multimodal=True  # This enables multimodal capabilities
)
```

When you set `multimodal=True`, the agent is automatically configured with the necessary tools for handling non-text content, including the `AddImageTool`.

Here's a complete example showing how to use a multimodal agent to analyze an image:

```
from crewai import Agent, Task, Crew

# Create a multimodal agent
image_analyst = Agent(
    role="Product Analyst",
    goal="Analyze product images and provide detailed descriptions",
    backstory="Expert in visual product analysis with deep knowledge of design an
    multimodal=True
)

# Create a task for image analysis
task = Task(
    description="Analyze the product image at https://example.com/product.jpg and
    expected_output="A detailed description of the product image",
    agent=image_analyst
)

# Create and run the crew
crew = Crew(
    agents=[image_analyst],
    tasks=[task]
)

result = crew.kickoff()
```

Advanced Usage with Context

You can provide additional context or specific questions about the image when creating tasks for multimodal agents. The task description can include specific aspects you want the

```
# Create a multimodal agent for detailed analysis
expert_analyst = Agent(
    role="Visual Quality Inspector",
    goal="Perform detailed quality analysis of product images",
    backstory="Senior quality control expert with expertise in visual inspection"
    multimodal=True  # AddImageTool is automatically included
)

# Create a task with specific analysis requirements
inspection_task = Task(
    description="""
    Analyze the product image at https://example.com/product.jpg with focus on:
    1. Quality of materials
    2. Manufacturing defects
    3. Compliance with standards
    Provide a detailed report highlighting any issues found.
    """,
    expected_output="A detailed report highlighting any issues found",
    agent=expert_analyst
)

# Create and run the crew
crew = Crew(
    agents=[expert_analyst],
    tasks=[inspection_task]
)

result = crew.kickoff()
```

Tool Details

When working with multimodal agents, the `AddImageTool` is automatically configured with the following schema:

The multimodal agent will automatically handle the image processing through its built-in tools, allowing it to:

- Access images via URLs or local file paths
- Process image content with optional context or specific questions
- Provide analysis and insights based on the visual information and task requirements

Best Practices

When working with multimodal agents, keep these best practices in mind:

1. **Image Access**
   - Ensure your images are accessible via URLs that the agent can reach
   - For local images, consider hosting them temporarily or using absolute file paths
   - Verify that image URLs are valid and accessible before running tasks
2. **Task Description**
   - Be specific about what aspects of the image you want the agent to analyze
   - Include clear questions or requirements in the task description
   - Consider using the optional `action` parameter for focused analysis
3. **Resource Management**
   - Image processing may require more computational resources than text-only tasks
   - Some language models may require base64 encoding for image data
   - Consider batch processing for multiple images to optimize performance

Test with small images first to validate your setup

5. Error Handling

Implement proper error handling for image loading failures

Have fallback strategies for when image processing fails

Monitor and log image processing operations for debugging

How to Guides

Coding Agents

Learn how to enable your CrewAI Agents to write and execute code, and explore advanced features for enhanced functionality.

Introduction

CrewAI Agents now have the powerful ability to write and execute code, significantly enhancing their problem-solving capabilities. This feature is particularly useful for tasks that require computational or programmatic solutions.

Enabling Code Execution

To enable code execution for an agent, set the `allow_code_execution` parameter to `True` when creating the agent.

Here's an example:

Code

```
from crewai import Agent

coding_agent = Agent(
    role="Senior Python Developer",
    goal="Craft well-designed and thought-out code",
    backstory="You are a senior Python developer with extensive experience in sof
    allow_code_execution=True
)
```

Important Considerations

1. **Model Selection**: It is strongly recommended to use more capable models like Claude 3.5 Sonnet and GPT-4 when enabling code execution. These models have a better understanding of programming concepts and are more likely to generate correct and efficient code.

2. **Error Handling**: The code execution feature includes error handling. If executed code raises an exception, the agent will receive the error message and can attempt to correct the code or provide alternative solutions. The `max_retry_limit` parameter, which defaults to 2, controls the maximum number of retries for a task.

3. **Dependencies**: To use the code execution feature, you need to install the `crewai_tools` package. If not installed, the agent will log an info message: "Coding tools not available. Install crewai_tools."

Code Execution Process

When an agent with code execution enabled encounters a task requiring programming:

1 **Task Analysis**

The agent analyzes the task and determines that code execution is necessary.

2 **Code Formulation**

It formulates the Python code needed to solve the problem.

4 **Result Interpretation**

The agent interprets the result and incorporates it into its response or uses it for further problem-solving.

Example Usage

Here's a detailed example of creating an agent with code execution capabilities and using it in a task:

Code

```
from crewai import Agent, Task, Crew

# Create an agent with code execution enabled
coding_agent = Agent(
    role="Python Data Analyst",
    goal="Analyze data and provide insights using Python",
    backstory="You are an experienced data analyst with strong Python skills.",
    allow_code_execution=True
)

# Create a task that requires code execution
data_analysis_task = Task(
    description="Analyze the given dataset and calculate the average age of parti
    agent=coding_agent
)

# Create a crew and add the task
analysis_crew = Crew(
    agents=[coding_agent],
    tasks=[data_analysis_task]
```

```
print(result)
```

In this example, the `coding_agent` can write and execute Python code to perform data analysis tasks.

‹ Using Multimodal Agents Force Tool Output as Result ›

How to Guides

Force Tool Output as Result

Learn how to force tool output as the result in an Agent's task in CrewAI.

Introduction

In CrewAI, you can force the output of a tool as the result of an agent's task. This feature is useful when you want to ensure that the tool output is captured and returned as the task result, avoiding any agent modification during the task execution.

Forcing Tool Output as Result

To force the tool output as the result of an agent's task, you need to set the `result_as_answer` parameter to `True` when adding a tool to the agent. This parameter ensures that the tool output is captured and returned as the task result, without any modifications by the agent.

Here's an example of how to force the tool output as the result of an agent's task:

Code

```
from crewai.agent import Agent
from my_tool import MyCustomTool

# Create a coding agent with the custom tool
coding_agent = Agent(
        role="Data Scientist",
        goal="Produce amazing reports on AI",
        backstory="You work with data and AI",
```

Workflow in Action

1 **Task Execution**

The agent executes the task using the tool provided.

2 **Tool Output**

The tool generates the output, which is captured as the task result.

3 **Agent Interaction**

The agent may reflect and take learnings from the tool but the output is not modified.

4 **Result Return**

The tool output is returned as the task result without any modifications.

‹ Coding Agents Human Input on Execution ›

How to Guides

Human Input on Execution

Integrating CrewAI with human input during execution in complex decision-making processes and leveraging the full capabilities of the agent's attributes and tools.

Human input in agent execution

Human input is critical in several agent execution scenarios, allowing agents to request additional information or clarification when necessary. This feature is especially useful in complex decision-making processes or when agents require more details to complete a task effectively.

Using human input with CrewAI

To integrate human input into agent execution, set the `human_input` flag in the task definition. When enabled, the agent prompts the user for input before delivering its final answer. This input can provide extra context, clarify ambiguities, or validate the agent's output.

Example:

```
pip install crewai
```

Code

```
os.environ["SERPER_API_KEY"] = "Your Key"  # serper.dev API key
os.environ["OPENAI_API_KEY"] = "Your Key"

# Loading Tools
search_tool = SerperDevTool()

# Define your agents with roles, goals, tools, and additional attributes
researcher = Agent(
    role='Senior Research Analyst',
    goal='Uncover cutting-edge developments in AI and data science',
    backstory=(
        "You are a Senior Research Analyst at a leading tech think tank. "
        "Your expertise lies in identifying emerging trends and technologies in A
        "You have a knack for dissecting complex data and presenting actionable i
    ),
    verbose=True,
    allow_delegation=False,
    tools=[search_tool]
)
writer = Agent(
    role='Tech Content Strategist',
    goal='Craft compelling content on tech advancements',
    backstory=(
        "You are a renowned Tech Content Strategist, known for your insightful an
        "With a deep understanding of the tech industry, you transform complex co
    ),
    verbose=True,
    allow_delegation=True,
    tools=[search_tool],
    cache=False,  # Disable cache for this agent
)

# Create tasks for your agents
task1 = Task(
    description=(
        "Conduct a comprehensive analysis of the latest advancements in AI in 202
```

```
    agent=researcher,
    human_input=True
)

task2 = Task(
    description=(
        "Using the insights from the researcher\'s report, develop an engaging bl
        "Your post should be informative yet accessible, catering to a tech-savvy
        "Aim for a narrative that captures the essence of these breakthroughs and
    ),
    expected_output='A compelling 3 paragraphs blog post formatted as markdown ab
    agent=writer,
    human_input=True
)

# Instantiate your crew with a sequential process
crew = Crew(
    agents=[researcher, writer],
    tasks=[task1, task2],
    verbose=True,
    memory=True,
    planning=True  # Enable planning feature for the crew
)

# Get your crew to work!
result = crew.kickoff()

print("######################")
print(result)
```

Was this page helpful? Yes No

How to Guides

Kickoff Crew Asynchronously

Kickoff a Crew Asynchronously

Introduction

CrewAI provides the ability to kickoff a crew asynchronously, allowing you to start the crew execution in a non-blocking manner. This feature is particularly useful when you want to run multiple crews concurrently or when you need to perform other tasks while the crew is executing.

Asynchronous Crew Execution

To kickoff a crew asynchronously, use the `kickoff_async()` method. This method initiates the crew execution in a separate thread, allowing the main thread to continue executing other tasks.

Method Signature

Code

```
def kickoff_async(self, inputs: dict) -> CrewOutput:
```

Parameters

`inputs` (dict): A dictionary containing the input data required for the tasks.

Potential Use Cases

Parallel Content Generation: Kickoff multiple independent crews asynchronously, each responsible for generating content on different topics. For example, one crew might research and draft an article on AI trends, while another crew generates social media posts about a new product launch. Each crew operates independently, allowing content production to scale efficiently.

Concurrent Market Research Tasks: Launch multiple crews asynchronously to conduct market research in parallel. One crew might analyze industry trends, while another examines competitor strategies, and yet another evaluates consumer sentiment. Each crew independently completes its task, enabling faster and more comprehensive insights.

Independent Travel Planning Modules: Execute separate crews to independently plan different aspects of a trip. One crew might handle flight options, another handles accommodation, and a third plans activities. Each crew works asynchronously, allowing various components of the trip to be planned simultaneously and independently for faster results.

Example: Single Asynchronous Crew Execution

Here's an example of how to kickoff a crew asynchronously using asyncio and awaiting the result:

Code

```
import asyncio
from crewai import Crew, Agent, Task

# Create an agent with code execution enabled
```

```
)

# Create a task that requires code execution
data_analysis_task = Task(
    description="Analyze the given dataset and calculate the average age of parti
    agent=coding_agent
)

# Create a crew and add the task
analysis_crew = Crew(
    agents=[coding_agent],
    tasks=[data_analysis_task]
)

# Async function to kickoff the crew asynchronously
async def async_crew_execution():
    result = await analysis_crew.kickoff_async(inputs={"ages": [25, 30, 35, 40, 4
    print("Crew Result:", result)

# Run the async function
asyncio.run(async_crew_execution())
```

Example: Multiple Asynchronous Crew Executions

In this example, we'll show how to kickoff multiple crews asynchronously and wait for all of them to complete using `asyncio.gather()`.

Code

```
import asyncio
from crewai import Crew, Agent, Task
```

```
    allow_code_execution=True
)

# Create tasks that require code execution
task_1 = Task(
    description="Analyze the first dataset and calculate the average age of parti
    agent=coding_agent
)

task_2 = Task(
    description="Analyze the second dataset and calculate the average age of part
    agent=coding_agent
)

# Create two crews and add tasks
crew_1 = Crew(agents=[coding_agent], tasks=[task_1])
crew_2 = Crew(agents=[coding_agent], tasks=[task_2])

# Async function to kickoff multiple crews asynchronously and wait for all to fin
async def async_multiple_crews():
    result_1 = crew_1.kickoff_async(inputs={"ages": [25, 30, 35, 40, 45]})
    result_2 = crew_2.kickoff_async(inputs={"ages": [20, 22, 24, 28, 30]})

    # Wait for both crews to finish
    results = await asyncio.gather(result_1, result_2)

    for i, result in enumerate(results, 1):
        print(f"Crew {i} Result:", result)

# Run the async function
asyncio.run(async_multiple_crews())
```

How to Guides

Kickoff Crew for Each

Kickoff Crew for Each Item in a List

Introduction

CrewAI provides the ability to kickoff a crew for each item in a list, allowing you to execute the crew for each item in the list. This feature is particularly useful when you need to perform the same set of tasks for multiple items.

Kicking Off a Crew for Each Item

To kickoff a crew for each item in a list, use the `kickoff_for_each()` method. This method executes the crew for each item in the list, allowing you to process multiple items efficiently.

Here's an example of how to kickoff a crew for each item in a list:

Code

```
from crewai import Crew, Agent, Task

# Create an agent with code execution enabled
coding_agent = Agent(
    role="Python Data Analyst",
    goal="Analyze data and provide insights using Python",
    backstory="You are an experienced data analyst with strong Python skills.",
    allow_code_execution=True
)

# Create a task that requires code execution
```

```
# Create a crew and add the task
analysis_crew = Crew(
    agents=[coding_agent],
    tasks=[data_analysis_task],
    verbose=True,
    memory=False,
    respect_context_window=True  # enable by default
)

datasets = [
  { "ages": [25, 30, 35, 40, 45] },
  { "ages": [20, 25, 30, 35, 40] },
  { "ages": [30, 35, 40, 45, 50] }
]

# Execute the crew
result = analysis_crew.kickoff_for_each(inputs=datasets)
```


‹ Kickoff Crew Asynchronously Replay Tasks from Latest Crew Kickoff ›

How to Guides

Replay Tasks from Latest Crew Kickoff

Replay tasks from the latest crew.kickoff(...)

Introduction

CrewAI provides the ability to replay from a task specified from the latest crew kickoff. This feature is particularly useful when you've finished a kickoff and may want to retry certain tasks or don't need to refetch data over and your agents already have the context saved from the kickoff execution so you just need to replay the tasks you want to.

> You must run `crew.kickoff()` before you can replay a task. Currently, only the latest kickoff is supported, so if you use `kickoff_for_each`, it will only allow you to replay from the most recent crew run.

Here's an example of how to replay from a task:

Replaying from Specific Task Using the CLI

To use the replay feature, follow these steps:

1. Open your terminal or command prompt.
2. Navigate to the directory where your CrewAI project is located.
3. Run the following commands:

Once you have your `task_id` to replay, use:

```
crewai replay -t <task_id>
```

Ensure `crewai` is installed and configured correctly in your development environment.

Replaying from a Task Programmatically

To replay from a task programmatically, use the following steps:

1 **Specify the `task_id` and input parameters for the replay process.**

Specify the `task_id` and input parameters for the replay process.

2 **Execute the replay command within a try-except block to handle potential errors.**

Execute the replay command within a try-except block to handle potential errors.

Code

```
def replay():
"""
Replay the crew execution from a specific task.
"""
task_id = '<task_id>'
inputs = {"topic": "CrewAI Training"}  # This is optional; you can pass
try:
```

```
    except Exception as e:
        raise Exception(f"An unexpected error occurred: {e}")
```

Conclusion

With the above enhancements and detailed functionality, replaying specific tasks in CrewAI has been made more efficient and robust. Ensure you follow the commands and steps precisely to make the most of these features.

‹ Kickoff Crew for Each Conditional Tasks ›

How to Guides

Conditional Tasks

Learn how to use conditional tasks in a crewAI kickoff

Introduction

Conditional Tasks in crewAI allow for dynamic workflow adaptation based on the outcomes of previous tasks. This powerful feature enables crews to make decisions and execute tasks selectively, enhancing the flexibility and efficiency of your AI-driven processes.

Example Usage

Code

```
from typing import List
from pydantic import BaseModel
from crewai import Agent, Crew
from crewai.tasks.conditional_task import ConditionalTask
from crewai.tasks.task_output import TaskOutput
from crewai.task import Task
from crewai_tools import SerperDevTool

# Define a condition function for the conditional task
# If false, the task will be skipped, if true, then execute the task.
def is_data_missing(output: TaskOutput) -> bool:
    return len(output.pydantic.events) < 10  # this will skip this task

# Define the agents
data_fetcher_agent = Agent(
    role="Data Fetcher",
```

```
data_processor_agent = Agent(
    role="Data Processor",
    goal="Process fetched data",
    backstory="Backstory 2",
    verbose=True
)

summary_generator_agent = Agent(
    role="Summary Generator",
    goal="Generate summary from fetched data",
    backstory="Backstory 3",
    verbose=True
)

class EventOutput(BaseModel):
    events: List[str]

task1 = Task(
    description="Fetch data about events in San Francisco using Serper tool",
    expected_output="List of 10 things to do in SF this week",
    agent=data_fetcher_agent,
    output_pydantic=EventOutput,
)

conditional_task = ConditionalTask(
    description="""
        Check if data is missing. If we have less than 10 events,
        fetch more events using Serper tool so that
        we have a total of 10 events in SF this week..
        """,
    expected_output="List of 10 Things to do in SF this week",
    condition=is_data_missing,
    agent=data_processor_agent,
)
```

```
)

# Create a crew with the tasks
crew = Crew(
    agents=[data_fetcher_agent, data_processor_agent, summary_generator_agent],
    tasks=[task1, conditional_task, task3],
    verbose=True,
    planning=True
)

# Run the crew
result = crew.kickoff()
print("results", result)
```


‹ Replay Tasks from Latest Crew Kickoff

Agent Monitoring with AgentOps ›

How to Guides

Agent Monitoring with AgentOps

Understanding and logging your agent performance with AgentOps.

Introduction

Observability is a key aspect of developing and deploying conversational AI agents. It allows developers to understand how their agents are performing, how their agents are interacting with users, and how their agents use external tools and APIs. AgentOps is a product independent of CrewAI that provides a comprehensive observability solution for agents.

AgentOps

AgentOps provides session replays, metrics, and monitoring for agents.

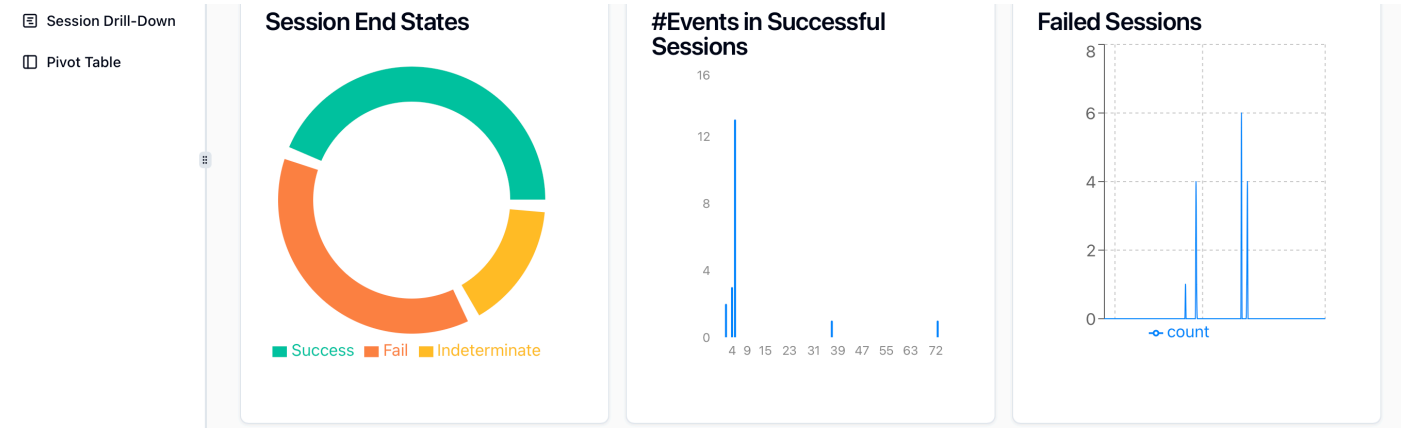
At a high level, AgentOps gives you the ability to monitor cost, token usage, latency, agent failures, session-wide statistics, and more. For more info, check out the AgentOps Repo.

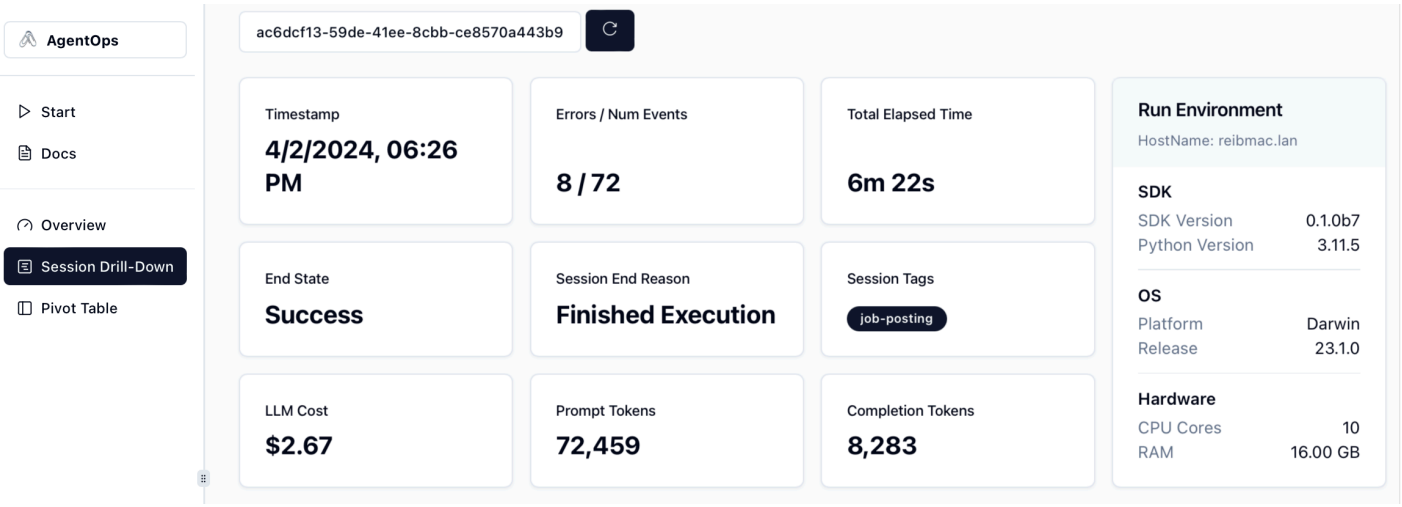
Overview

AgentOps provides monitoring for agents in development and production. It provides a dashboard for tracking agent performance, session replays, and custom reporting.

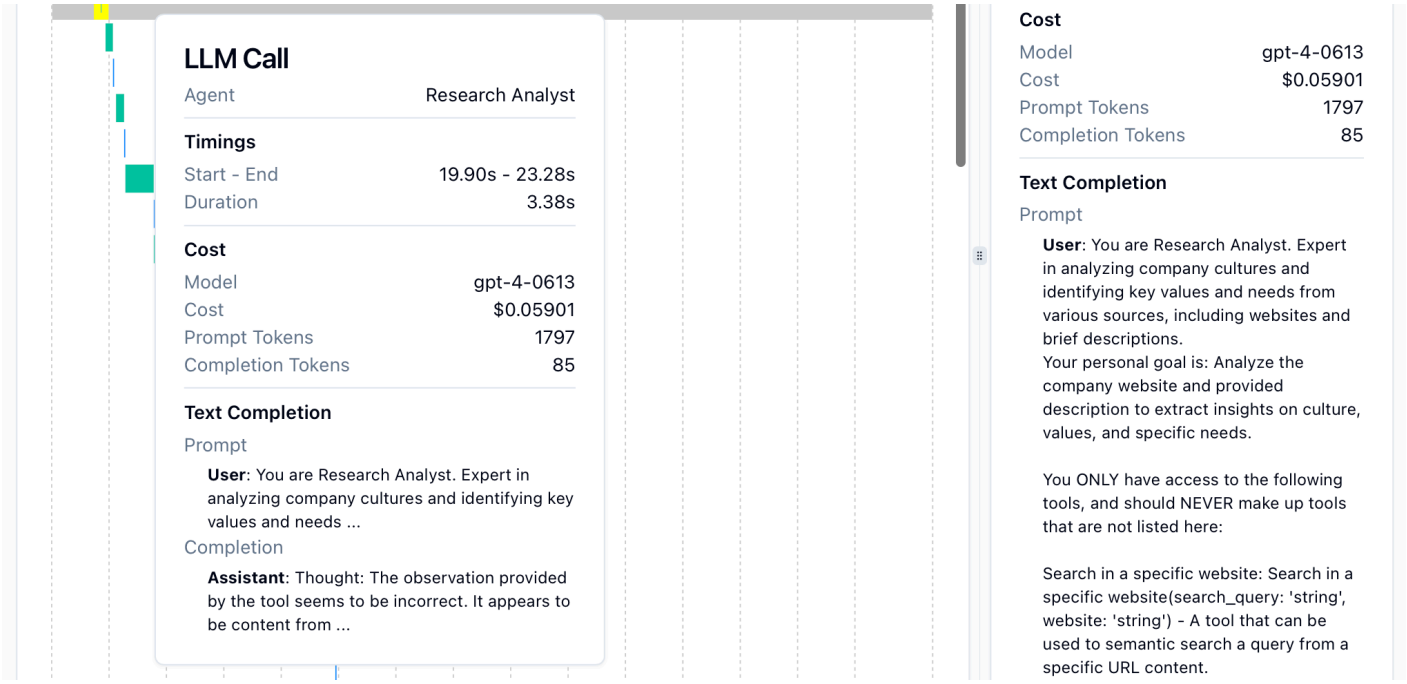
Additionally, AgentOps provides session drilldowns for viewing Crew agent interactions, LLM calls, and tool usage in real-time. This feature is useful for debugging and understanding how agents interact with users as well as other agents.

How to Guides › Agent Monitoring with AgentOps





How to Guides › Agent Monitoring with AgentOps



Features

LLM Cost Management and Tracking: Track spend with foundation model providers.

Replay Analytics: Watch step-by-step agent execution graphs.

Recursive Thought Detection: Identify when agents fall into infinite loops.

Custom Reporting: Create custom analytics on agent performance.

Analytics Dashboard: Monitor high-level statistics about agents in development and production.

Public Model Testing: Test your agents against benchmarks and leaderboards.

Custom Tests: Run your agents against domain-specific tests.

Time Travel Debugging: Restart your sessions from checkpoints.

Compliance and Security: Create audit logs and detect potential threats such as profanity and PII leaks.

Prompt Injection Detection: Identify potential code injection and secret leaks.

Create a user API key here: Create API Key

2 Configure Your Environment

Add your API key to your environment variables:

```
AGENTOPS_API_KEY=<YOUR_AGENTOPS_API_KEY>
```

3 Install AgentOps

Install AgentOps with:

```
pip install 'crewai[agentops]'
```

or

```
pip install agentops
```

4 Initialize AgentOps

Before using `Crew` in your script, include these lines:

```
import agentops
agentops.init()
```

Crew + AgentOps Examples

Job Posting

Example of a Crew agent that generates job posts.

Markdown Validator

Example of a Crew agent that validates Markdown files.

Instagram Post

Example of a Crew agent that generates Instagram posts.

Further Information

To get started, create an AgentOps account.

For feature requests or bug reports, please reach out to the AgentOps team on the AgentOps Repo.

Extra links

Twitter • Discord • AgentOps Dashboard • Documentation

How to Guides

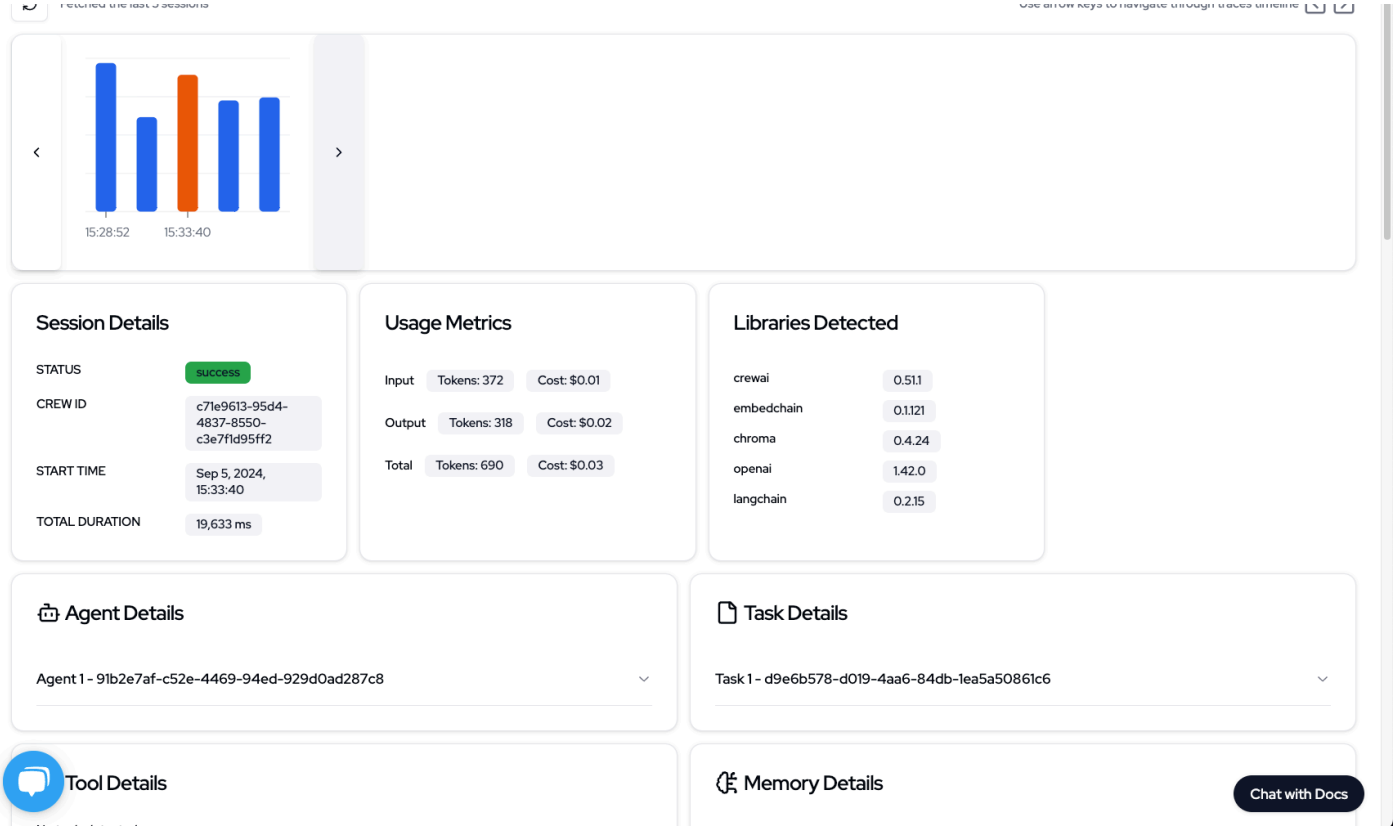
Agent Monitoring with Langtrace

How to monitor cost, latency, and performance of CrewAI Agents using Langtrace, an external observability tool.

Langtrace Overview

Langtrace is an open-source, external tool that helps you set up observability and evaluations for Large Language Models (LLMs), LLM frameworks, and Vector Databases. While not built directly into CrewAI, Langtrace can be used alongside CrewAI to gain deep visibility into the cost, latency, and performance of your CrewAI Agents. This integration allows you to log hyperparameters, monitor performance regressions, and establish a process for continuous improvement of your Agents.

How to Guides › Agent Monitoring with Langtrace



How to Guides › Agent Monitoring with Langtrace

How to Guides › Agent Monitoring with Langtrace

Setup Instructions

1 Sign up for Langtrace

Sign up by visiting https://langtrace.ai/signup.

2 Create a project

Set the project type to `CrewAI` and generate an API key.

3 Install Langtrace in your CrewAI project

Use the following command:

Import and initialize Langtrace at the beginning of your script, before any CrewAI imports:

```
from langtrace_python_sdk import langtrace
langtrace.init(api_key='<LANGTRACE_API_KEY>')

# Now import CrewAI modules
from crewai import Agent, Task, Crew
```

Features and Their Application to CrewAI

1. **LLM Token and Cost Tracking**

   Monitor the token usage and associated costs for each CrewAI agent interaction.

2. **Trace Graph for Execution Steps**

   Visualize the execution flow of your CrewAI tasks, including latency and logs.

   Useful for identifying bottlenecks in your agent workflows.

3. **Dataset Curation with Manual Annotation**

   Create datasets from your CrewAI task outputs for future training or evaluation.

4. **Prompt Versioning and Management**

   Keep track of different versions of prompts used in your CrewAI agents.

   Useful for A/B testing and optimizing agent performance.

6. Testing and Evaluations

Set up automated tests for your CrewAI agents and tasks.

‹ Agent Monitoring with AgentOps

Agent Monitoring with MLflow ›

How to Guides

Agent Monitoring with MLflow

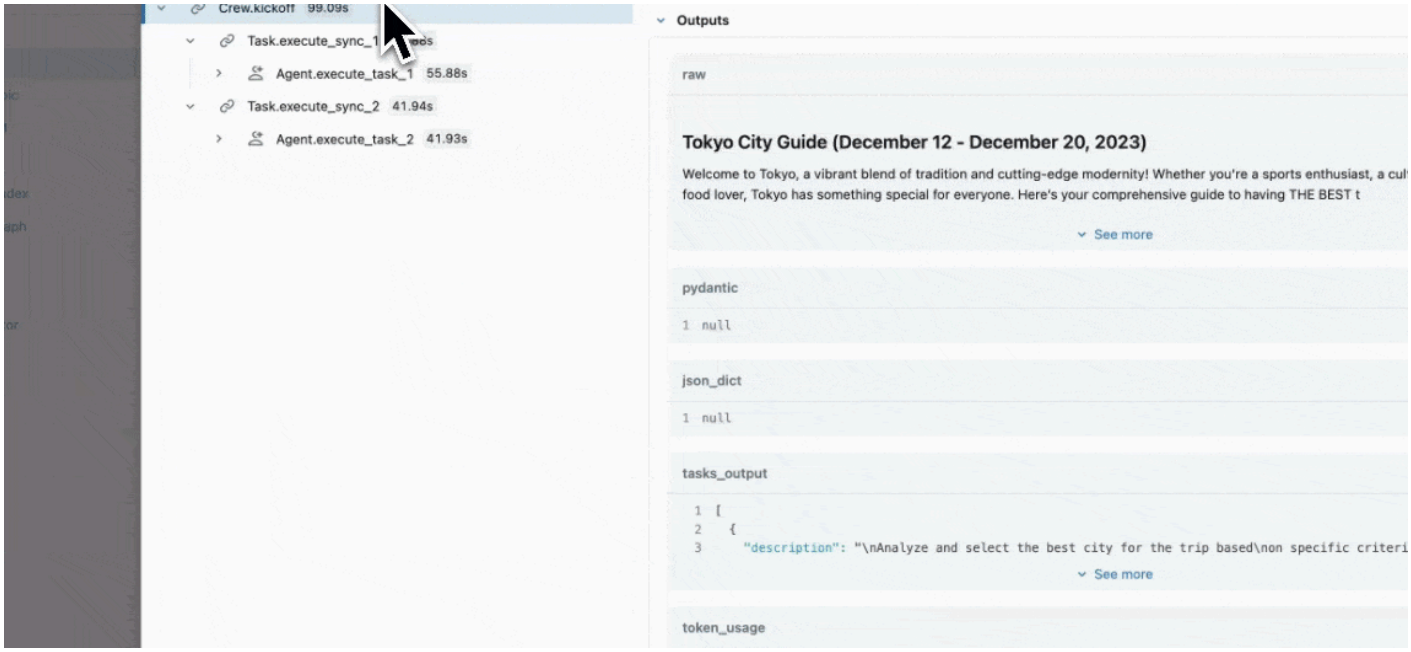
Quickly start monitoring your Agents with MLflow.

MLflow Overview

MLflow is an open-source platform to assist machine learning practitioners and teams in handling the complexities of the machine learning process.

It provides a tracing feature that enhances LLM observability in your Generative AI applications by capturing detailed information about the execution of your application's services. Tracing provides a way to record the inputs, outputs, and metadata associated with each intermediate step of a request, enabling you to easily pinpoint the source of bugs and unexpected behaviors.

How to Guides › Agent Monitoring with MLflow



Features

Tracing Dashboard: Monitor activities of your crewAI agents with detailed dashboards that include inputs, outputs and metadata of spans.

Automated Tracing: A fully automated integration with crewAI, which can be enabled by running `mlflow.crewai.autolog()` .

Manual Trace Instrumentation with minor efforts: Customize trace instrumentation through MLflow's high-level fluent APIs such as decorators, function wrappers and context managers.

OpenTelemetry Compatibility: MLflow Tracing supports exporting traces to an OpenTelemetry Collector, which can then be used to export traces to various backends such as Jaeger, Zipkin, and AWS X-Ray.

Package and Deploy Agents: Package and deploy your crewAI agents to an inference server with a variety of deployment targets.

Securely Host LLMs: Host multiple LLM from various providers in one unified endpoint through MFflow gateway.

Setup Instructions

1 Install MLflow package

```
# The crewAI integration is available in mlflow>=2.19.0
pip install mlflow
```

2 Start MFflow tracking server

```
# This process is optional, but it is recommended to use MLflow track  g s
mlflow server
```

3 Initialize MLflow in Your Application

Add the following two lines to your application code:

```
import mlflow

mlflow.crewai.autolog()

# Optional: Set a tracking URI and an experiment name if you have a tracki
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("CrewAI")
```

Example Usage for tracing CrewAI Agents:

```
from textwrap import dedent

content = "Users name is John. He is 30 years old and lives in San Francis
string_source = StringKnowledgeSource(
    content=content, metadata={"preference": "personal"}
)

search_tool = WebsiteSearchTool()


class TripAgents:
    def city_selection_agent(self):
        return Agent(
            role="City Selection Expert",
            goal="Select the best city based on weather, season, and price
            backstory="An expert in analyzing travel data to pick ideal de
            tools=[
                search_tool,
            ],
            verbose=True,
        )

    def local_expert(self):
        return Agent(
            role="Local Expert at this city",
            goal="Provide the BEST insights about the selected city",
            backstory="""A knowledgeable local guide with extensive inform
        about the city, it's attractions and customs""",
            tools=[search_tool],
            verbose=True,
        )


class TripTasks:
    def identify_task(self, agent, origin, cities, interests, range):
        return Task(
```

```
                multiple cities, considering factors like current weather
                conditions, upcoming cultural or seasonal events, and
                overall travel expenses.
                Your final answer must be a detailed
                report on the chosen city, and everything you found out
                about it, including the actual flight costs, weather
                forecast and attractions.

                Traveling from: {origin}
                City Options: {cities}
                Trip Date: {range}
                Traveler Interests: {interests}
            """
            ),
            agent=agent,
            expected_output="Detailed report on the chosen city including
        )

    def gather_task(self, agent, origin, interests, range):
        return Task(
            description=dedent(
                f"""
                As a local expert on this city you must compile an
                in-depth guide for someone traveling there and wanting
                to have THE BEST trip ever!
                Gather information about key attractions, local customs,
                special events, and daily activity recommendations.
                Find the best spots to go to, the kind of place only a
                local would know.
                This guide should provide a thorough overview of what
                the city has to offer, including hidden gems, cultural
                hotspots, must-visit landmarks, weather forecasts, and
                high level costs.
                The final answer must be a comprehensive city guide,
                rich in cultural insights and practical tips,
```

```
            """
            ),
            agent=agent,
            expected_output="Comprehensive city guide including hidden gem
        )


class TripCrew:
    def __init__(self, origin, cities, date_range, interests):
        self.cities = cities
        self.origin = origin
        self.interests = interests
        self.date_range = date_range

    def run(self):
        agents = TripAgents()
        tasks = TripTasks()

        city_selector_agent = agents.city_selection_agent()
        local_expert_agent = agents.local_expert()

        identify_task = tasks.identify_task(
            city_selector_agent,
            self.origin,
            self.cities,
            self.interests,
            self.date_range,
        )
        gather_task = tasks.gather_task(
            local_expert_agent, self.origin, self.interests, self.date_ran
        )

        crew = Crew(
            agents=[city_selector_agent, local_expert_agent],
            tasks=[identify_task, gather_task],
```

```
            },
        )

        result = crew.kickoff()
        return result


trip_crew = TripCrew("California", "Tokyo", "Dec 12 - Dec 20", "sports")
result = trip_crew.run()

print(result)
```

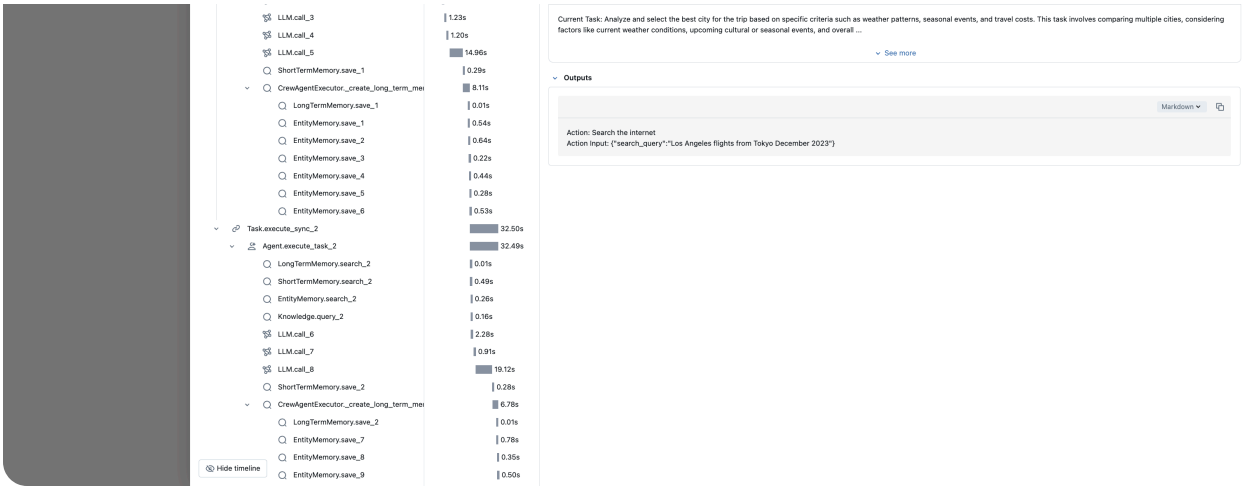
Refer to MLflow Tracing Documentation for more configurations and use cases.

4 Visualize Activities of Agents

Now traces for your crewAI agents are captured by MLflow. Let's visit MLflow tracking server to view the traces and get insights into your Agents.

Open `127.0.0.1:5000` on your browser to visit MLflow tracking server.

How to Guides › Agent Monitoring with MLflow



MLflow Tracing Dashboard

Was this page helpful? Yes No

‹ Agent Monitoring with Langtrace

Agent Monitoring with OpenLIT ›

Powered by Mintlify

How to Guides

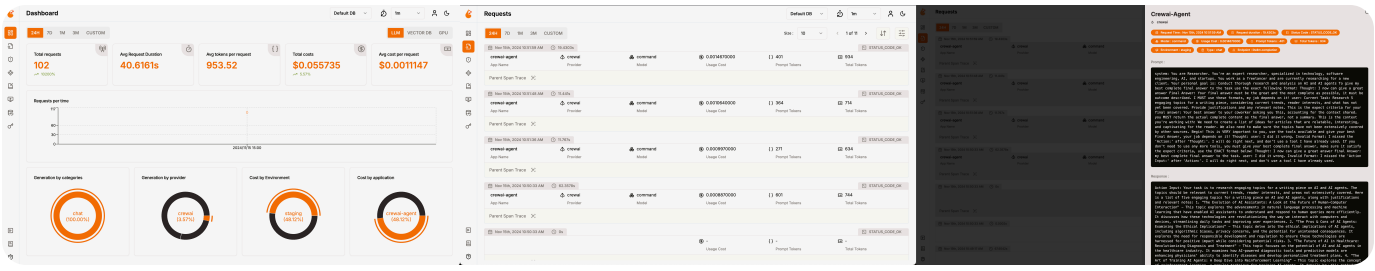
Agent Monitoring with OpenLIT

Quickly start monitoring your Agents in just a single line of code with OpenTelemetry.

OpenLIT Overview

OpenLIT is an open-source tool that makes it simple to monitor the performance of AI agents, LLMs, VectorDBs, and GPUs with just one line of code.

It provides OpenTelemetry-native tracing and metrics to track important parameters like cost, latency, interactions and task sequences. This setup enables you to track hyperparameters and monitor for performance issues, helping you find ways to enhance and fine-tune your agents over time.



OpenLIT Dashboard

Features

- Analytics Dashboard: Monitor your Agents health and performance with detailed dashboards that track metrics, costs, and user interactions.

Exceptions Monitoring Dashboard: Quickly spot and resolve issues by tracking common exceptions and errors with a monitoring dashboard.

Compliance and Security: Detect potential threats such as profanity and PII leaks.

Prompt Injection Detection: Identify potential code injection and secret leaks.

API Keys and Secrets Management: Securely handle your LLM API keys and secrets centrally, avoiding insecure practices.

Prompt Management: Manage and version Agent prompts using PromptHub for consistent and easy access across Agents.

Model Playground Test and compare different models for your CrewAI agents before deployment.

Setup Instructions

1 Deploy OpenLIT

1 Git Clone OpenLIT Repository

```
git clone git@github.com:openlit/openlit.git
```

```
docker compose up -d
```

2 Install OpenLIT SDK

```
pip install openlit
```

3 Initialize OpenLIT in Your Application

Add the following two lines to your application code:

Setup using function arguments | Setup using Environment Variables

```
import openlit
openlit.init(otlp_endpoint="http://127.0.0.1:4318")
```

Example Usage for monitoring a CrewAI Agent:

```
from crewai import Agent, Task, Crew, Process
import openlit

openlit.init(disable_metrics=True)
# Define your agents
researcher = Agent(
    role="Researcher",
    goal="Conduct thorough research and analysis on AI and AI agents",
    backstory="You're an expert researcher, specialized in technology, sof
```

```
# Define your task
task = Task(
    description="Generate a list of 5 interesting ideas for an article, th
    expected_output="5 bullet points, each with a paragraph and accompanyi
)

# Define the manager agent
manager = Agent(
    role="Project Manager",
    goal="Efficiently manage the crew and ensure high-quality task complet
    backstory="You're an experienced project manager, skilled in overseeir
    allow_delegation=True,
    llm='command-r'
)

# Instantiate your crew with a custom manager
crew = Crew(
    agents=[researcher],
    tasks=[task],
    manager_agent=manager,
    process=Process.hierarchical,
)

# Start the crew's work
result = crew.kickoff()

print(result)
```

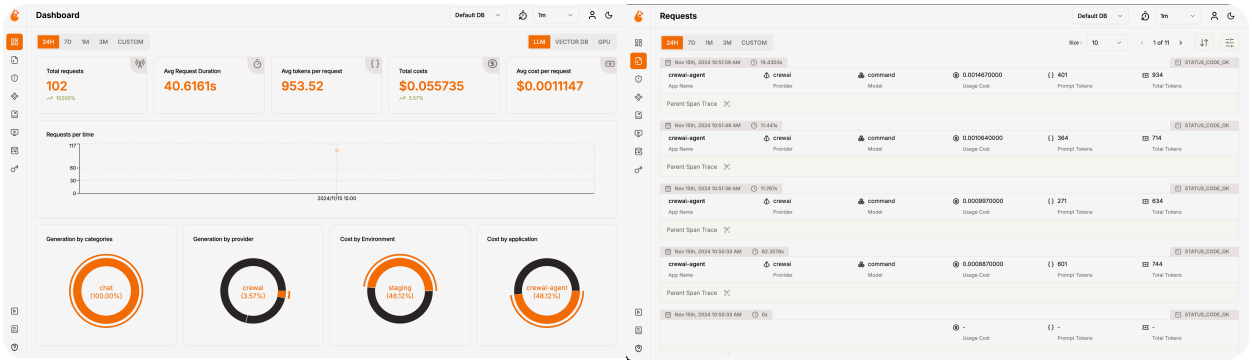
Refer to OpenLIT Python SDK repository for more advanced configurations and use cases.

4 Visualize and Analyze

Just head over to OpenLIT at `127.0.0.1:3000` on your browser to start exploring. You can login using the default credentials

Email: `user@openlit.io`

Password: `openlituser`



OpenLIT Dashboard

Was this page helpful? Yes No

‹ Agent Monitoring with MLflow

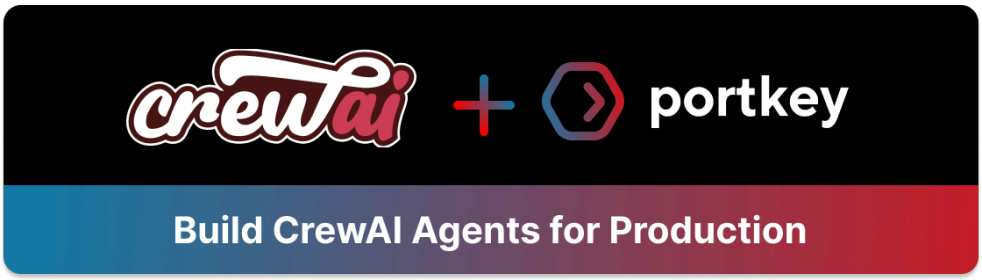
Agent Monitoring with Portkey ›

Powered by Mintlify

How to Guides

Agent Monitoring with Portkey

How to use Portkey with CrewAI



Portkey is a 2-line upgrade to make your CrewAI agents reliable, cost-efficient, and fast.

Portkey adds 4 core production capabilities to any CrewAI agent:

1. Routing to 200+ LLMs
2. Making each LLM call more robust
3. Full-stack tracing & cost, performance analytics
4. Real-time guardrails to enforce behavior

Getting Started

1 Install CrewAI and Portkey

```
pip install -qU crewai portkey-ai
```

Portkey API Key: Sign up on the Portkey app and copy your API Key

Virtual Key: Virtual Keys securely manage your LLM API keys in one place. Store your LLM provider API keys securely in Portkey's vault

```
from crewai import LLM
from portkey_ai import createHeaders, PORTKEY_GATEWAY_URL

gpt_llm = LLM(
    model="gpt-4",
    base_url=PORTKEY_GATEWAY_URL,
    api_key="dummy", # We are using Virtual key
    extra_headers=createHeaders(
        api_key="YOUR_PORTKEY_API_KEY",
        virtual_key="YOUR_VIRTUAL_KEY", # Enter your Virtual key from Port
    )
)
```

3 Create and Run Your First Agent

```
from crewai import Agent, Task, Crew

# Define your agents with roles and goals
coder = Agent(
    role='Software developer',
    goal='Write clear, concise code on demand',
    backstory='An expert coder with a keen eye for software trends.',
    llm=gpt_llm
)

# Create tasks for your agents
task1 = Task(
    description="Define the HTML for making a simple website with heading-
```

```
crew = Crew(
    agents=[coder],
    tasks=[task1],
)

result = crew.kickoff()
print(result)
```

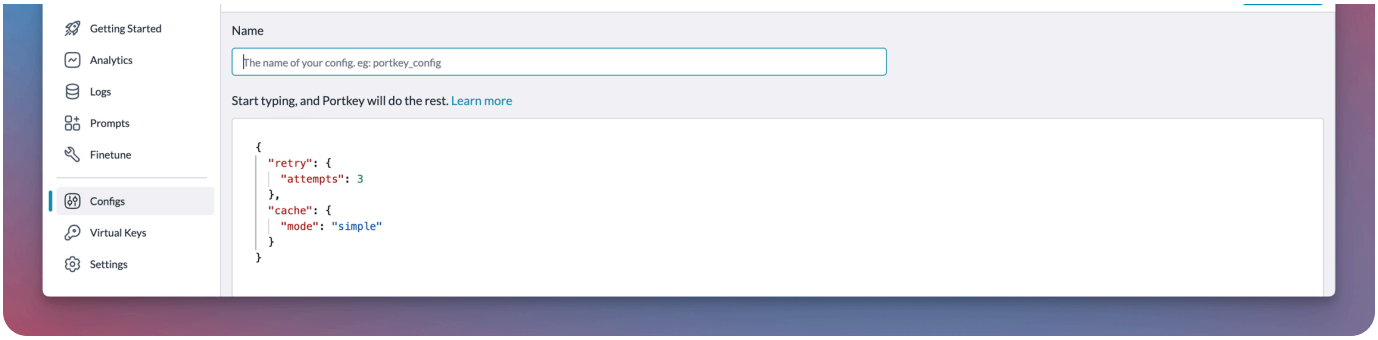
Key Features

Feature	Description
🌐 Multi-LLM Support	Access OpenAI, Anthropic, Gemini, Azure, and 250+ providers through a unified interface
🛡️ Production Reliability	Implement retries, timeouts, load balancing, and fallbacks
📊 Advanced Observability	Track 40+ metrics including costs, tokens, latency, and custom metadata
🔍 Comprehensive Logging	Debug with detailed execution traces and function call logs
🚧 Security Controls	Set budget limits and implement role-based access control
🔄 Performance Analytics	Capture and analyze feedback for continuous improvement
💾 Intelligent Caching	Reduce costs and latency with semantic or simple caching

Production Features with Portkey Configs

All features mentioned below are through Portkey's Config system. Portkey's Config system allows you to define routing strategies using simple JSON objects in your LLM API calls. You

How to Guides › Agent Monitoring with Portkey



1. Use 250+ LLMs

Access various LLMs like Anthropic, Gemini, Mistral, Azure OpenAI, and more with minimal code changes. Switch between providers or use them together seamlessly. Learn more about Universal API

Easily switch between different LLM providers:

```
# Anthropic Configuration
anthropic_llm = LLM(
    model="claude-3-5-sonnet-latest",
    base_url=PORTKEY_GATEWAY_URL,
    api_key="dummy",
    extra_headers=createHeaders(
        api_key="YOUR_PORTKEY_API_KEY",
        virtual_key="YOUR_ANTHROPIC_VIRTUAL_KEY", #You don't need provider when u
        trace_id="anthropic_agent"
    )
)

# Azure OpenAI Configuration
azure_llm = LLM(
    model="gpt-4",
    base_url=PORTKEY_GATEWAY_URL,
    api_key="dummy",
```

```
)
```

2. Caching

Improve response times and reduce costs with two powerful caching modes:

- **Simple Cache**: Perfect for exact matches
- **Semantic Cache**: Matches responses for requests that are semantically similar Learn more about Caching

```
config = {
    "cache": {
        "mode": "semantic",  # or "simple" for exact matching
    }
}
```

3. Production Reliability

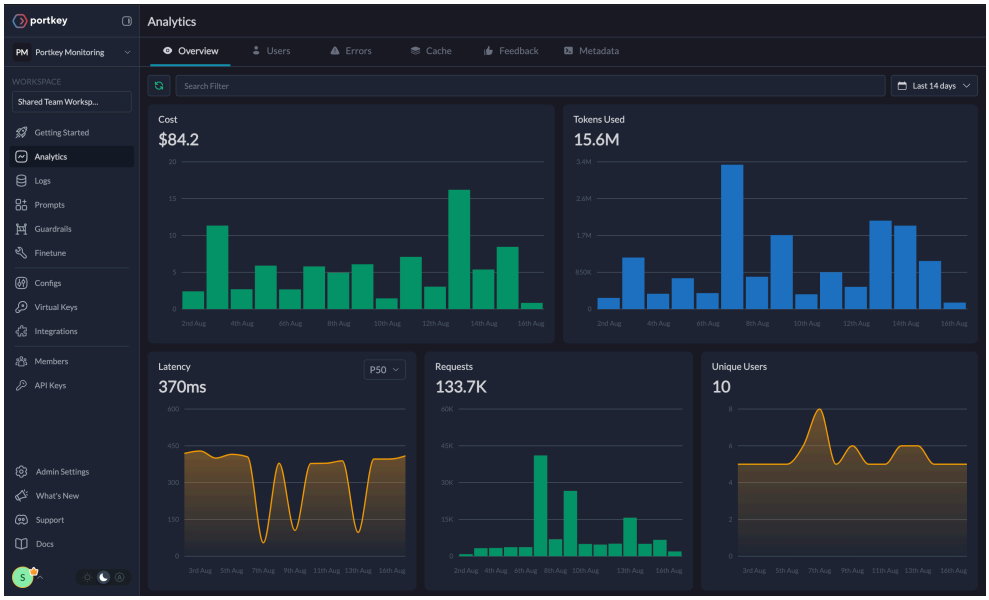
Portkey provides comprehensive reliability features:

- **Automatic Retries**: Handle temporary failures gracefully
- **Request Timeouts**: Prevent hanging operations
- **Conditional Routing**: Route requests based on specific conditions
- **Fallbacks**: Set up automatic provider failovers
- **Load Balancing**: Distribute requests efficiently

Learn more about Reliability Features

need.

- Cost per agent interaction
- Response times and latency
- Token usage and efficiency
- Success/failure rates
- Cache hit rates



5. Detailed Logging

Logs are essential for understanding agent behavior, diagnosing issues, and improving performance. They provide a detailed record of agent activities and tool use, which is crucial for debugging and optimizing processes.

Access a dedicated section to view records of agent executions, including parameters, outcomes, function calls, and errors. Filter logs based on multiple parameters such as trace ID, model, tokens used, and metadata.

Track system changes with audit logs

Configure data retention policies

For detailed information on creating and managing Configs, visit the Portkey documentation.

Resources

- 📘 Portkey Documentation
- 📊 Portkey Dashboard
- 🐦 Twitter
- 💬 Discord Community

Was this page helpful? Yes No

‹ Agent Monitoring with OpenLIT

Agent Monitoring with Langfuse ›

How to Guides

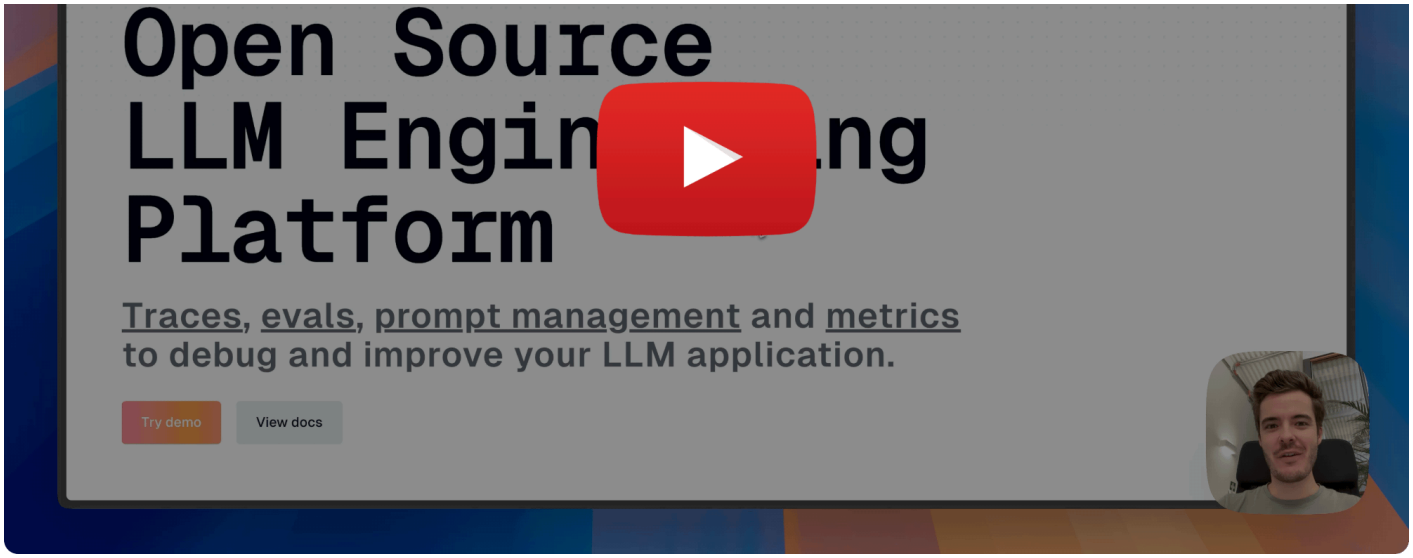
Agent Monitoring with Langfuse

Learn how to integrate Langfuse with CrewAI via OpenTelemetry using OpenLit

Integrate Langfuse with CrewAI

This notebook demonstrates how to integrate **Langfuse** with **CrewAI** using OpenTelemetry via the **OpenLit** SDK. By the end of this notebook, you will be able to trace your CrewAI applications with Langfuse for improved observability and debugging.

> **What is Langfuse?** Langfuse is an open-source LLM engineering platform. It provides tracing and monitoring capabilities for LLM applications, helping developers debug, analyze, and optimize their AI systems. Langfuse integrates with various tools and frameworks via native integrations, OpenTelemetry, and APIs/SDKs.



Get Started

We'll walk through a simple example of using CrewAI and integrating it with Langfuse via OpenTelemetry using OpenLit.

Step 1: Install Dependencies

```
%pip install langfuse openlit crewai crewai_tools
```

Step 2: Set Up Environment Variables

Set your Langfuse API keys and configure OpenTelemetry export settings to send traces to Langfuse. Please refer to the Langfuse OpenTelemetry Docs for more information on the Langfuse OpenTelemetry endpoint `/api/public/otel` and authentication.

```
import os
import base64
```

```
os.environ["OTEL_EXPORTER_OTLP_ENDPOINT"] = "https://cloud.langfuse.com/api/publi
# os.environ["OTEL_EXPORTER_OTLP_ENDPOINT"] = "https://us.cloud.langfuse.com/api/
os.environ["OTEL_EXPORTER_OTLP_HEADERS"] = f"Authorization=Basic {LANGFUSE_AUTH}"

# your openai key
os.environ["OPENAI_API_KEY"] = "sk-..."
```

Step 3: Initialize OpenLit

Initialize the OpenLit OpenTelemetry instrumentation SDK to start capturing OpenTelemetry traces.

```
import openlit

openlit.init()
```

Step 4: Create a Simple CrewAI Application

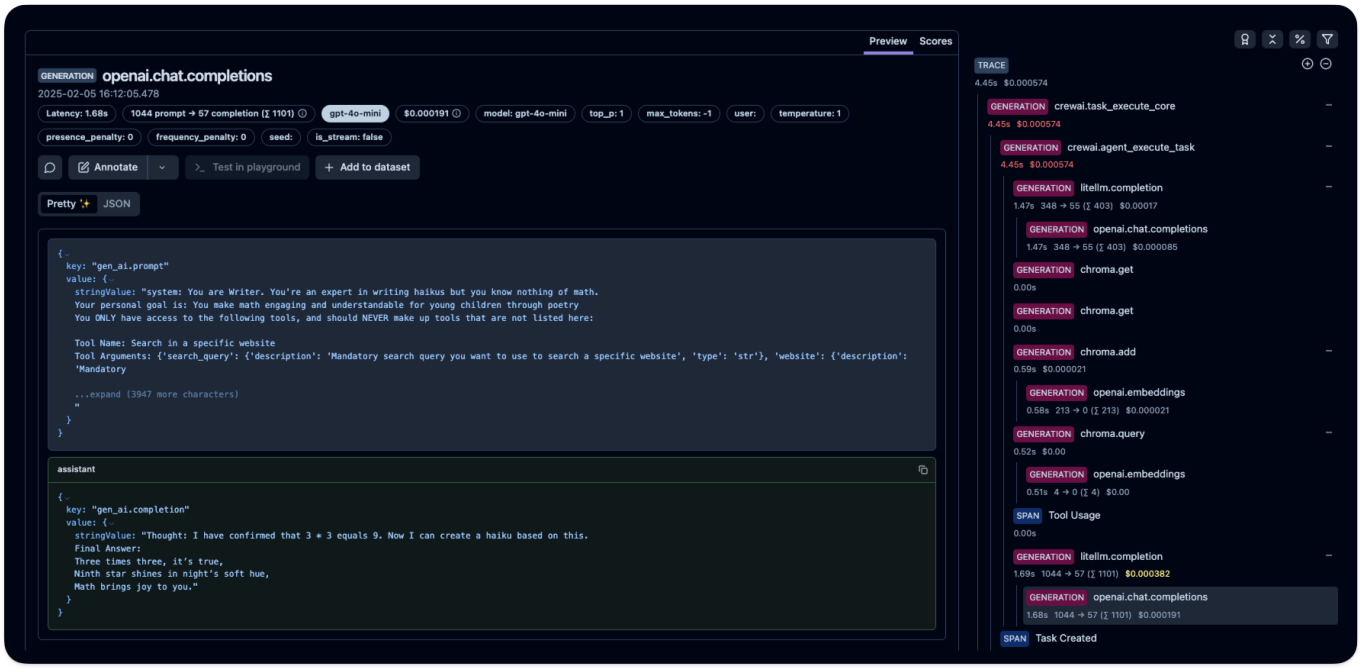
We'll create a simple CrewAI application where multiple agents collaborate to answer a user's question.

```
from crewai import Agent, Task, Crew

from crewai_tools import (
    WebsiteSearchTool
)

web_rag_tool = WebsiteSearchTool()

writer = Agent(
```

```
task = Task(description=("What is {multiplication}?"),
            expected_output=("Compose a haiku that includes the answer."),
            agent=writer)

crew = Crew(
  agents=[writer],
  tasks=[task],
  share_crew=False
)
```

Step 5: See Traces in Langfuse

After running the agent, you can view the traces generated by your CrewAI application in Langfuse. You should see detailed steps of the LLM interactions, which can help you debug and optimize your AI agent.



Langfuse OpenTelemetry Docs

Was this page helpful? Yes No

‹ Agent Monitoring with Portkey

Browserbase Web Loader ›