CrewAI Documentation

Agents
What is an Agent?¶
What is an Agent?

An agent is an autonomous unit programmed to:

Perform tasks
Make decisions
Communicate with other agents

Think of an agent as a member of a team, with specific skills and a particular job to do. Agents can have different roles like 'Researcher', 'Writer', or 'Customer Support', each contributing to the overall goal of the crew.
Agent Attributes¶
Attribute Description
Role Defines the agent's function within the crew. It determines the kind of tasks the agent is best suited for.
Goal The individual objective that the agent aims to achieve. It guides the agent's decision-making process.
Backstory Provides context to the agent's role and goal, enriching the interaction and collaboration dynamics.
LLM (optional) The language model used by the agent to process and generate text. It dynamically fetches the model name from the OPENAI_MODEL_NAME environment variable, defaulting to "gpt-4" if not specified.
Tools (optional) Set of capabilities or functions that the agent can use to perform tasks. Tools can be shared or exclusive to specific agents. It's an attribute that can be set during the initialization of an agent, with a default value of an empty list.
Function Calling LLM (optional) If passed, this agent will use this LLM to execute function calling for tools instead of relying on the main LLM output.
Max Iter (optional) The maximum number of iterations the agent can perform before being forced to give its best answer. Default is 15.
Max RPM (optional) The maximum number of requests per minute the agent can perform to avoid rate limits. It's optional and can be left unspecified, with a default value of None.
Verbose (optional) Enables detailed logging of the agent's execution for debugging or monitoring purposes when set to True. Default is False.
Allow Delegation (optional) Agents can delegate tasks or questions to one another, ensuring that each task is handled by the most suitable agent. Default is True.
Step Callback (optional) A function that is called after each step of the agent. This can be used to log the agent's actions or to perform other operations. It will overwrite the crew step_callback.
Memory (optional) Indicates whether the agent should have memory or not, with a default value of False. This impacts the agent's ability to remember past interactions. Default is False.

Creating an Agent¶
Agent Interaction

Agents can interact with each other using crewAI's built-in delegation and communication mechanisms. This allows for dynamic task management and problem-solving within the crew.

To create an agent, you would typically initialize an instance of the Agent class with the desired properties. Here's a conceptual example including all attributes:

```
# Example: Creating an agent with all attributes
from crewai import Agent

agent = Agent(
  role='Data Analyst',
  goal='Extract actionable insights',
  backstory="""You're a data analyst at a large company.
  You're responsible for analyzing data and providing insights
  to the business.
  You're currently working on a project to analyze the
  performance of our marketing campaigns.""",
  tools=[my_tool1, my_tool2],  # Optional, defaults to an empty list
  llm=my_llm,  # Optional
  function_calling_llm=my_llm,  # Optional
  max_iter=15,  # Optional
  max_rpm=None, # Optional
  verbose=True,  # Optional
  allow_delegation=True,  # Optional
  step_callback=my_intermediate_step_callback,  # Optional
  memory=True  # Optional
)
```

Conclusion¶
Agents are the building blocks of the CrewAI framework. By understanding how to define and interact with agents, you can create sophisticated AI systems that leverage the power of collaborative intelligence.

Tasks
Overview of a Task¶
What is a Task?

In the CrewAI framework, tasks are individual assignments that agents complete. They encapsulate necessary information for execution, including a description, assigned agent, required tools, offering flexibility for various action complexities.

Tasks in CrewAI can be designed to require collaboration between agents. For example, one agent might gather data while another analyzes it. This collaborative approach can be defined within the task properties and managed by the Crew's process.

Task Attributes¶
Attribute Description
Description A clear, concise statement of what the task entails.
Agent Optionally, you can specify which agent is responsible for the task. If not, the crew's process will determine who takes it on.
Expected Output Clear and detailed definition of expected output for the task.
Tools (optional) These are the functions or capabilities the agent can utilize to perform the task. They can be anything from simple actions like 'search' to more complex interactions with other agents or APIs.
Async Execution (optional) Indicates whether the task should be executed asynchronously, allowing the crew to continue with the next task without waiting for completion.
Context (optional) Other tasks that will have their output used as context for this task. If a task is asynchronous, the system will wait for that to finish before using its output as context.
Output JSON (optional) Takes a pydantic model and returns the output as a JSON object. Agent LLM needs to be using an OpenAI client, could be Ollama for example but using the OpenAI wrapper
Output Pydantic (optional) Takes a pydantic model and returns the output as a pydantic object. Agent LLM needs to be using an OpenAI client, could be Ollama for example but using the OpenAI wrapper
Output File (optional) Takes a file path and saves the output of the task on it.
Callback (optional) A function to be executed after the task is completed.
Creating a Task¶
This is the simplest example for creating a task, it involves defining its scope and agent, but there are optional attributes that can provide a lot of flexibility:

```
from crewai import Task

task = Task(
    description='Find and summarize the latest and most relevant news on AI',
    agent=sales_agent
)
```

Task Assignment

Tasks can be assigned directly by specifying an agent to them, or they can be assigned in run time if you are using the hierarchical through CrewAI's process, considering roles, availability, or other criteria.

Integrating Tools with Tasks¶

Tools from the crewAI Toolkit and LangChain Tools enhance task performance, allowing agents to interact more effectively with their environment. Assigning specific tools to tasks can tailor agent capabilities to particular needs.

Creating a Task with Tools¶

```
import os
os.environ["OPENAI_API_KEY"] = "Your Key"
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
)

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=2
)

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

This demonstrates how tasks with specific tools can override an agent's default set for tailored task execution.

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:

```
# ...

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

research_ops_task = Task(
  description='Find and summarize the latest AI Ops news',
  expected_output='A bullet list summary of the top 5 most important AI Ops news',
  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 news",
  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.

```
#...
```

```
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 events.",
    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 the two tasks to be
completed
)

#...
```

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.

```
# ...

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_output}
    """)

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:

```
# ...
task1 = 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=[task1, task2, task3],
    verbose=2
)

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_output}
""")
```

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:

Ensuring only one output type is set per task to maintain clear output expectations.

Preventing the manual assignment of the id attribute to uphold the integrity of the unique identifier system.
These validations help in maintaining the consistency and reliability of task executions within the crewAI framework.

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.

Tools
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?¶
Definition

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.
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:

```
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(
    role='Market Research Analyst',
    goal='Provide up-to-date market analysis of the AI industry',
    backstory='An expert analyst with a keen eye for market trends.',
    tools=[search_tool, web_rag_tool],
    verbose=True
)

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 summary.',
    expected_output='A summary of the top 3 trending developments in the AI industry with a
unique perspective on their significance.',
    agent=researcher
)

write = Task(
```

```
    description='Write an engaging blog post about the AI industry, based on the research analyst's summary. Draw inspiration from the latest blog posts in the directory.',
    expected_output='A 4-paragraph blog post formatted in markdown with engaging, informative, and accessible content, avoiding complex jargon.',
    agent=writer,
    output_file='blog-posts/new_post.md'  # The final blog post will be saved here
)

# Assemble a crew
crew = Crew(
    agents=[researcher, writer],
    tasks=[research, write],
    verbose=2
)

# Execute tasks
crew.kickoff()
```

Available crewAI Tools¶

Most of the tools in the crewAI toolkit offer the ability to set specific arguments or let them to be more wide open, this is the case for most of the tools, for example:

```
from crewai_tools import DirectoryReadTool

# This will allow the agent with this tool to read any directory it wants during it's execution
tool = DirectoryReadTool()

# OR

# This will allow the agent with this tool to read only the directory specified during it's execution
toos = DirectoryReadTool(directory='./directory')
```

Specific per tool docs are coming soon. Here is a list of the available tools and their descriptions:

Tool	Description
CodeDocsSearchTool	A RAG tool optimized for searching through code documentation and related technical documents.
CSVSearchTool	A RAG tool designed for searching within CSV files, tailored to handle structured data.
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.
FileReadTool Enables reading and extracting data from files, supporting various file formats.
GithubSearchTool A RAG tool for searching within GitHub repositories, useful for code and documentation search.
SeperDevTool 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.
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.
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.
Creating your own Tools¶
Custom Tool Creation

Developers can craft custom tools tailored for their agent's needs or utilize pre-built options:

To create your own crewAI tools you will need to install our extra tools package:

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

Once you do that there are two main ways for one to create a crewAI tool:

Subclassing BaseTool¶

```
from crewai_tools import BaseTool
```

```
class MyCustomTool(BaseTool):
    name: str = "Name of my tool"
    description: str = "Clear description for what this tool is useful for, you agent will need this information to use it."

    def _run(self, argument: str) -> str:
        # Implementation goes here
        return "Result from custom tool"
```

Define a new class inheriting from BaseTool, specifying name, description, and the _run method for operational logic.

Utilizing the tool Decorator¶

For a simpler approach, create a Tool object directly with the required attributes and a functional logic.

```
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, you agent will need this information to use it."""
    # Function logic here
```

```
import json
import requests
from crewai import Agent
from crewai.tools import tool
from unstructured.partition.html import partition_html

    # Annotate the function with the tool decorator from crewAI
@tool("Integration with a given API")
def integration_tool(argument: str) -> str:
    """Integration with a given API"""
    # Code here
    return resutls # string to be sent back to the agent

# Assign the scraping tool 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=[integration_tool]
)
```

Using LangChain Tools¶
LangChain Integration

CrewAI seamlessly integrates with LangChain’s comprehensive toolkit for search-based queries and more, here are the available built-in tools that are offered by Langchain LangChain Toolkit

:

```
from crewai import Agent
from langchain.agents import Tool
from langchain.utilities import GoogleSerperAPIWrapper

# Setup API keys
os.environ["SERPER_API_KEY"] = "Your Key"

search = GoogleSerperAPIWrapper()

# Create and assign the search tool to an agent
serper_tool = Tool(
  name="Intermediate Answer",
  func=search.run,
  description="Useful for search-based queries",
)

agent = Agent(
  role='Research Analyst',
  goal='Provide up-to-date market analysis',
  backstory='An expert analyst with a keen eye for market trends.',
  tools=[serper_tool]
)

# 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.

Processes
Understanding Processes¶
Core Concept

In CrewAI, 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. Note: A manager language model (manager_llm) must be specified in the crew to enable the hierarchical process, allowing for the creation and management of tasks by the manager.
Consensual Process (Planned): Currently under consideration for future development, this process type aims for collaborative decision-making among agents on task execution, introducing a more democratic approach to task management within CrewAI. As of now, it is not implemented in the codebase.
The Role of Processes in Teamwork¶
Processes enable individual agents to operate as a cohesive unit, streamlining their efforts to achieve common objectives with efficiency and coherence.

Assigning Processes to a Crew¶
To assign a process to a crew, specify the process type upon crew creation to set the execution strategy. Note: For a hierarchical process, ensure to define manager_llm for the manager agent.

```
from crewai import Crew
from crewai.process import Process
from langchain_openai import ChatOpenAI

# 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
crew = Crew(
    agents=my_agents,
    tasks=my_tasks,
    process=Process.hierarchical,
    manager_llm=ChatOpenAI(model="gpt-4")
)
```

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

Sequential Process¶

This method mirrors dynamic team workflows, progressing through tasks in a thoughtful and systematic manner. Task execution follows the predefined order in the task list, with the output of one task serving as context for the next.

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 creates a manager automatically for you, requiring the specification of a manager language model (manager_llm) for the manager agent. 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, and future consensual). This design choice guarantees that only valid processes are utilized within the CrewAI framework.

Planned Future Processes¶

Consensual Process: This collaborative decision-making process among agents on task execution is under consideration but not currently implemented. This future enhancement aims to introduce a more democratic approach to task management within CrewAI.

Conclusion¶

The structured collaboration facilitated by processes within CrewAI is crucial for enabling systematic teamwork among agents. Documentation will be regularly updated to reflect new processes and enhancements, ensuring users have access to the most current and comprehensive information.

Crews

What is a Crew?¶

Definition of 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 Description

Tasks A list of tasks assigned to the crew.
Agents A list of agents that are part of the crew.
Process (optional) The process flow (e.g., sequential, hierarchical) the crew follows.
Verbose (optional) The verbosity level for logging during execution.
Manager LLM (optional) The language model used by the manager agent in a hierarchical process. Required when using a hierarchical process.
Function Calling LLM (optional) 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.
Config (optional) Optional configuration settings for the crew, in Json or Dict[str, Any] format.
Max RPM (optional) Maximum requests per minute the crew adheres to during execution.
Language (optional) Language used for the crew, defaults to English.
Full Output (optional) Whether the crew should return the full output with all tasks outputs or just the final output.
Step Callback (optional) 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.
Share Crew (optional) 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.
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 a Crew¶
Crew Composition

When assembling a crew, you combine agents with complementary roles and tools, assign tasks, and select a process that dictates their execution order and interaction.

Example: Assembling a Crew¶

```
from crewai import Crew, Agent, Task, Process
from langchain_community.tools import DuckDuckGoSearchRun

# Define agents with specific roles and tools
researcher = Agent(
    role='Senior Research Analyst',
    goal='Discover innovative AI technologies',
    tools=[DuckDuckGoSearchRun()]
)

writer = Agent(
```

```
    role='Content Writer',
    goal='Write engaging articles on AI discoveries',
    verbose=True
)

# Create tasks for the agents
research_task = Task(
    description='Identify breakthrough AI technologies',
    agent=researcher
)
write_article_task = Task(
    description='Draft an article on the latest AI technologies',
    agent=writer
)

# Assemble the crew with a sequential process
my_crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, write_article_task],
    process=Process.sequential,
    full_output=True,
    verbose=True,
)
```

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.

```
# 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 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.

```
# Start the crew's task execution
```

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

Collaboration
Collaboration Fundamentals¶
Core of Agent Interaction

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:

Language Model Management (manager_llm, function_calling_llm): Manages language models for executing tasks and tools, facilitating sophisticated agent-tool interactions. It's important to note that manager_llm is mandatory when using a hierarchical process for ensuring proper execution flow.
Process Flow (process): Defines the execution logic (e.g., sequential, hierarchical) to streamline task distribution and execution.
Verbose Logging (verbose): Offers detailed logging capabilities for monitoring and debugging purposes. It supports both integer and boolean types to indicate the verbosity level.
Configuration (config): Allows extensive customization to tailor the crew's behavior according to specific requirements.
Rate Limiting (max_rpm): Ensures efficient utilization of resources by limiting requests per minute.
Internationalization Support (language): Facilitates operation in multiple languages, enhancing global usability.
Execution and Output Handling (full_output): Distinguishes between full and final outputs for nuanced control over task results.
Callback and Telemetry (step_callback): Integrates callbacks for step-wise execution monitoring and telemetry for performance analytics.
Crew Sharing (share_crew): Enables sharing of crew information with CrewAI for continuous improvement and training models.
Usage Metrics (usage_metrics): Store all metrics for the language model (LLM) usage during all tasks' execution, providing insights into operational efficiency and areas for improvement, you can check it after the crew execution.
Delegation: Dividing to Conquer¶

Delegation enhances functionality by allowing agents to intelligently assign tasks or seek help, thereby amplifying the crew's overall capability.

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.

Getting Started
Introduction¶
Embark on your CrewAI journey by setting up your environment and initiating your AI crew with enhanced features. This guide ensures a seamless start, incorporating the latest updates.

Step 0: Installation¶
Install CrewAI and any necessary packages for your project.

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

Step 1: Assemble Your Agents¶
Define your agents with distinct roles, backstories, and now, enhanced capabilities such as verbose mode and memory usage. These elements add depth and guide their task execution and interaction within the crew.

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

```
from crewai import Agent
from crewai_tools import SerperDevTool
search_tool = SerperDevTool()

# Creating a senior researcher agent with memory and verbose mode
researcher = Agent(
  role='Senior Researcher',
  goal='Uncover groundbreaking technologies in {topic}',
  verbose=True,
  memory=True,
  backstory=(
    "Driven by curiosity, you're at the forefront of"
    "innovation, eager to explore and share knowledge that could change"
    "the world."
  ),
  tools=[search_tool],
  allow_delegation=True
)

# Creating a writer agent with custom tools and delegation capability
writer = Agent(
  role='Writer',
  goal='Narrate compelling tech stories about {topic}',
  verbose=True,
  memory=True,
  backstory=(
    "With a flair for simplifying complex topics, you craft"
    "engaging narratives that captivate and educate, bringing new"
    "discoveries to light in an accessible manner."
  ),
  tools=[search_tool],
  allow_delegation=False
)
```

Step 2: Define the Tasks¶

Detail the specific objectives for your agents, including new features for asynchronous execution and output customization. These tasks ensure a targeted approach to their roles.

```
from crewai import Task

# Research task
research_task = Task(
  description=(
    "Identify the next big trend in {topic}."
```

```
    "Focus on identifying pros and cons and the overall narrative."
    "Your final report should clearly articulate the key points"
    "its market opportunities, and potential risks."
  ),
  expected_output='A comprehensive 3 paragraphs long report on the latest AI trends.',
  tools=[search_tool],
  agent=researcher,
)

# Writing task with language model configuration
write_task = Task(
  description=(
    "Compose an insightful article on {topic}."
    "Focus on the latest trends and how it's impacting the industry."
    "This article should be easy to understand, engaging, and positive."
  ),
  expected_output='A 4 paragraph article on {topic} advancements formatted as markdown.',
  tools=[search_tool],
  agent=writer,
  async_execution=False,
  output_file='new-blog-post.md'  # Example of output customization
)
```

Step 3: Form the Crew¶

Combine your agents into a crew, setting the workflow process they'll follow to accomplish the tasks, now with the option to configure language models for enhanced interaction.

```
from crewai import Crew, Process

# Forming the tech-focused crew with enhanced configurations
crew = Crew(
  agents=[researcher, writer],
  tasks=[research_task, write_task],
  process=Process.sequential  # Optional: Sequential task execution is default
)
```

Step 4: Kick It Off¶

Initiate the process with your enhanced crew ready. Observe as your agents collaborate, leveraging their new capabilities for a successful project outcome. You can also pass the inputs that will be interpolated into the agents and tasks.

```
# Starting the task execution process with enhanced feedback
result = crew.kickoff(inputs={'topic': 'AI in healthcare'})
print(result)
```

Conclusion¶
Building and activating a crew in CrewAI has evolved with new functionalities. By incorporating verbose mode, memory capabilities, asynchronous task execution, output customization, and language model configuration, your AI team is more equipped than ever to tackle challenges efficiently. The depth of agent backstories and the precision of their objectives enrich collaboration, leading to successful project outcomes.

Create Custom Tools
Creating your own Tools¶
Custom Tool Creation

Developers can craft custom tools tailored for their agent's needs or utilize pre-built options:

To create your own crewAI tools you will need to install our extra tools package:

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

Once you do that there are two main ways for one to create a crewAI tool:

Subclassing BaseTool¶

```
from crewai_tools import BaseTool

class MyCustomTool(BaseTool):
    name: str = "Name of my tool"
    description: str = "Clear description for what this tool is useful for, you agent will need this information to use it."

    def _run(self, argument: str) -> str:
        # Implementation goes here
        return "Result from custom tool"
```

Define a new class inheriting from BaseTool, specifying name, description, and the _run method for operational logic.

Utilizing the tool Decorator¶
For a simpler approach, create a Tool object directly with the required attributes and a functional logic.

```
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, you agent will need this information to use it."""
    # Function logic here

import json
import requests
from crewai import Agent
from crewai.tools import tool
from unstructured.partition.html import partition_html

    # Annotate the function with the tool decorator from crewAI
@tool("Integration with a given API")
def integtation_tool(argument: str) -> str:
    """Integration with a given API"""
    # Code here
    return resutls # string to be sent back to the agent

# Assign the scraping tool 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=[integtation_tool]
)
```

Using the Tool function from langchain¶

For another simple approach, create a function in python directly with the required attributes and a functional logic.

```
def combine(a, b):
    return a + b
```

Then you can add that function into the your tool by using 'func' variable in the Tool function.

```
from langchain.agents import Tool

math_tool = Tool(
            name="Math tool",
            func=math_tool,
            description="Useful for adding two numbers together, in other words combining them."
        )
```

Using Sequential Process

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¶
Assemble your crew and define tasks in the order they need to be executed.

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

# 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 the tasks in sequence
research_task = Task(description='Gather relevant data...', agent=researcher)
analysis_task = Task(description='Analyze the data...', agent=analyst)
writing_task = Task(description='Compose the report...', agent=writer)

# Form the crew with a sequential process
report_crew = Crew(
  agents=[researcher, analyst, writer],
```

```
  tasks=[research_task, analysis_task, writing_task],
  process=Process.sequential
)
```

Workflow in Action¶

Initial Task: In a sequential process, the first agent completes their task and signals completion.
Subsequent Tasks: Agents pick up their tasks based on the process type, with outcomes of preceding tasks or manager directives guiding their execution.
Completion: The process concludes once the final task is executed, leading to project completion.

Conclusion¶

The sequential process in CrewAI provides a clear, straightforward path for task execution. It's particularly suited for projects requiring a logical progression of tasks, ensuring each step is completed before the next begins, thereby facilitating a cohesive final product.

Using Hierarchical Process

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.

Complexity and Efficiency

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 is automatically created by crewAI so you don't need to worry about it.

Key Features¶

Task Delegation: A manager agent allocates tasks among crew members based on their roles and capabilities.
Result Validation: The manager evaluates outcomes to ensure they meet the required standards.
Efficient Workflow: Emulates corporate structures, providing an organized approach to task management.

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.

Tools and Agent Assignment

Assign tools at the agent level to facilitate task delegation and execution by the designated agents under the manager's guidance.

Manager LLM Requirement

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.

```
from langchain_openai import ChatOpenAI
from crewai import Crew, Process, Agent

# Agents are defined with an optional tools parameter
researcher = Agent(
    role='Researcher',
    goal='Conduct in-depth analysis',
    # tools=[]  # This can be optionally specified; defaults to an empty list
)
writer = Agent(
    role='Writer',
    goal='Create engaging content',
    # tools=[]  # Optionally specify tools; defaults to an empty list
)

# Establishing the crew with a hierarchical process
project_crew = Crew(
    tasks=[...],  # Tasks to be delegated and executed under the manager's supervision
    agents=[researcher, writer],
    manager_llm=ChatOpenAI(temperature=0, model="gpt-4"),  # Mandatory for hierarchical
process
    process=Process.hierarchical  # Specifies the hierarchical management approach
)
```

Workflow in Action¶

Task Assignment: The manager assigns tasks strategically, considering each agent's capabilities.

Execution and Review: Agents complete their tasks, with the manager ensuring quality standards.

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.

Connecting to any LLM
Connect CrewAI to LLMs¶
Default LLM

By default, CrewAI uses OpenAI's GPT-4 model for language processing. However, you can configure your agents to use a different model or API. This guide will show you how to connect your agents to different LLMs through environment variables and direct instantiation.

CrewAI offers flexibility in connecting to various LLMs, including local models via Ollama and different APIs like Azure. It's compatible with all LangChain LLM components, enabling diverse integrations for tailored AI solutions.

CrewAI Agent Overview¶
The Agent class is the cornerstone for implementing AI solutions in CrewAI. Here's an updated overview reflecting the latest codebase changes:

Attributes:
role: Defines the agent's role within the solution.
goal: Specifies the agent's objective.
backstory: Provides a background story to the agent.
llm: Indicates the Large Language Model the agent uses.
function_calling_llm Optinal: Will turn the ReAct crewAI agent into a function calling agent.
max_iter: Maximum number of iterations for an agent to execute a task, default is 15.
memory: Enables the agent to retain information during the execution.
max_rpm: Sets the maximum number of requests per minute.
verbose: Enables detailed logging of the agent's execution.
allow_delegation: Allows the agent to delegate tasks to other agents, default is True.
tools: Specifies the tools available to the agent for task execution.
step_callback: Provides a callback function to be executed after each step.

```
# Required
os.environ["OPENAI_MODEL_NAME"]="gpt-4-0125-preview"

# Agent will automatically use the model defined in the environment variable
example_agent = Agent(
  role='Local Expert',
```

```
  goal='Provide insights about the city',
  backstory="A knowledgeable local guide.",
  verbose=True
)
```

Ollama Integration¶

Ollama is preferred for local LLM integration, offering customization and privacy benefits. To integrate Ollama with CrewAI, set the appropriate environment variables as shown below. Note: Detailed Ollama setup is beyond this document's scope, but general guidance is provided.

Setting Up Ollama¶

Environment Variables Configuration: To integrate Ollama, set the following environment variables:

```
OPENAI_API_BASE='http://localhost:11434/v1'
OPENAI_MODEL_NAME='openhermes'  # Adjust based on available model
OPENAI_API_KEY=''
```

OpenAI Compatible API Endpoints¶

Switch between APIs and models seamlessly using environment variables, supporting platforms like FastChat, LM Studio, and Mistral AI.

Configuration Examples¶

FastChat¶

```
OPENAI_API_BASE="http://localhost:8001/v1"
OPENAI_MODEL_NAME='oh-2.5m7b-q51'
OPENAI_API_KEY=NA
```

LM Studio¶

```
OPENAI_API_BASE="http://localhost:8000/v1"
OPENAI_MODEL_NAME=NA
OPENAI_API_KEY=NA
```

Mistral API¶

```
OPENAI_API_KEY=your-mistral-api-key
OPENAI_API_BASE=https://api.mistral.ai/v1
OPENAI_MODEL_NAME="mistral-small"
```

Azure Open AI Configuration¶

For Azure OpenAI API integration, set the following environment variables:

```
AZURE_OPENAI_VERSION="2022-12-01"
AZURE_OPENAI_DEPLOYMENT=""
AZURE_OPENAI_ENDPOINT=""
AZURE_OPENAI_KEY=""
```

Example Agent with Azure LLM¶

```
from dotenv import load_dotenv
from crewai import Agent
from langchain_openai import AzureChatOpenAI

load_dotenv()

azure_llm = AzureChatOpenAI(
    azure_endpoint=os.environ.get("AZURE_OPENAI_ENDPOINT"),
    api_key=os.environ.get("AZURE_OPENAI_KEY")
)

azure_agent = Agent(
  role='Example Agent',
  goal='Demonstrate custom LLM configuration',
  backstory='A diligent explorer of GitHub docs.',
  llm=azure_llm
)
```

Conclusion¶

Integrating CrewAI with different LLMs expands the framework's versatility, allowing for customized, efficient AI solutions across various domains and platforms.

Customizing Agents

Customizable Attributes¶

Crafting an efficient CrewAI team hinges on the ability to tailor your AI agents dynamically to meet the unique requirements of any project. This section covers the foundational attributes you can customize.

Key Attributes for Customization¶

Role: Specifies the agent's job within the crew, such as 'Analyst' or 'Customer Service Rep'.
Goal: Defines what the agent aims to achieve, in alignment with its role and the overarching objectives of the crew.
Backstory: Provides depth to the agent's persona, enriching its motivations and engagements within the crew.
Tools: Represents the capabilities or methods the agent uses to perform tasks, from simple functions to intricate integrations.

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. By default crewAI agents are ReAct agents, but by setting the function_calling_llm you can turn them into a function calling agents.

Enabling Memory for Agents¶
CrewAI supports memory for agents, enabling them to remember past interactions. This feature is critical for tasks requiring awareness of previous contexts or decisions.

Performance and Debugging Settings¶
Adjusting an agent's performance and monitoring its operations are crucial for efficient task execution.

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), controlling the agent's query frequency to external services.
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 15, providing a balance between thoroughness and efficiency. Once the agent approaches this number it will try it's 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. In this example we will use the crewAI tools package to create a tool for a research analyst agent.

pip install 'crewai[tools]'
Example: Assigning Tools to an Agent¶

```
import os
from crewai import 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,
  verbose=True,
  max_rpm=10, # Optional: Limit requests to 10 per minute, preventing API abuse
  max_iter=5, # Optional: Limit task iterations to 5 before the agent tries to give its best answer
  allow_delegation=False
)
```

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 set to True, enabling agents to seek assistance or delegate tasks as needed. This default behavior promotes collaborative problem-solving and efficiency within the crewAI ecosystem.

Example: Disabling Delegation for an Agent¶

```
agent = Agent(
  role='Content Writer',
  goal='Write engaging content on market trends',
  backstory='A seasoned writer with expertise in market analysis.',
  allow_delegation=False
)
```

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.

Human Input in Agent Execution¶

Human input plays a pivotal role in several agent execution scenarios, enabling agents to seek additional information or clarification when necessary. This capability is invaluable in complex decision-making processes or when agents need more details to complete a task effectively.

Using Human Input with CrewAI¶

Incorporating human input with CrewAI is straightforward, enhancing the agent's ability to make informed decisions. While the documentation previously mentioned using a "LangChain Tool" and a specific "DuckDuckGoSearchRun" tool from langchain_community.tools, it's important to

clarify that the integration of such tools should align with the actual capabilities and configurations defined within your Agent class setup.

Example:¶

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

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

from langchain.agents import load_tools

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


# Loading Human Tools
human_tools = load_tools(["human"])
search_tool = SerperDevTool()

# Define your agents with roles, goals, and tools
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 AI and data science."
    "You have a knack for dissecting complex data and presenting actionable insights."
  ),
  verbose=True,
  allow_delegation=False,
  tools=[search_tool]+human_tools # Passing human tools to the agent
)
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 and engaging articles on technology and innovation."
    "With a deep understanding of the tech industry, you transform complex concepts into compelling narratives."
  ),
```

```
  verbose=True,
  allow_delegation=True
)

# Create tasks for your agents
task1 = Task(
  description=(
    "Conduct a comprehensive analysis of the latest advancements in AI in 2024."
    "Identify key trends, breakthrough technologies, and potential industry impacts."
    "Compile your findings in a detailed report."
    "Make sure to check with a human if the draft is good before finalizing your answer."
  ),
  expected_output='A comprehensive full report on the latest AI advancements in 2024, leave
nothing out',
  agent=researcher,
)

task2 = Task(
  description=(
    "Using the insights from the researcher's report, develop an engaging blog post that highlights
the most significant AI advancements."
    "Your post should be informative yet accessible, catering to a tech-savvy audience."
    "Aim for a narrative that captures the essence of these breakthroughs and their implications
for the future."
  ),
  expected_output='A compelling 3 paragraphs blog post formatted as markdown about the
latest AI advancements in 2024',
  agent=writer
)

# Instantiate your crew with a sequential process
crew = Crew(
  agents=[researcher, writer],
  tasks=[task1, task2],
  verbose=2
)

# Get your crew to work!
result = crew.kickoff()

print("######################")
print(result)
```

SerperDevTool Documentation¶
Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is designed to perform a semantic search for a specified query from a text's content across the internet. It utilizes the serper.dev API to fetch and display the most relevant search results based on the query provided by the user.

Installation¶
To incorporate this tool into your project, follow the installation instructions below:

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

Example¶
The following example demonstrates how to initialize the tool and execute a search with a given query:

```
from crewai_tools import SerperDevTool

# Initialize the tool for internet searching capabilities
tool = SerperDevTool()
```

Steps to Get Started¶
To effectively use the SerperDevTool, follow these steps:

Package Installation: Confirm that the crewai[tools] package is installed in your Python environment.
API Key Acquisition: Acquire a serper.dev API key by registering for a free account at serper.dev.
Environment Configuration: Store your obtained API key in an environment variable named SERPER_API_KEY to facilitate its use by the tool.
Conclusion¶
By integrating the SerperDevTool into Python projects, users gain the ability to conduct real-time, relevant searches across the internet directly from their applications. By adhering to the setup and usage guidelines provided, incorporating this tool into projects is streamlined and straightforward.

ScrapeWebsiteTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
A tool designed to extract and read the content of a specified website. It is capable of handling various types of web pages by making HTTP requests and parsing the received HTML content. This tool can be particularly useful for web scraping tasks, data collection, or extracting specific information from websites.

Installation¶
Install the crewai_tools package

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

Example¶

```
from crewai_tools import ScrapeWebsiteTool

# To enable scrapping any website it finds during it's execution
tool = ScrapeWebsiteTool()

# Initialize the tool with the website URL, so the agent can only scrap the content of the specified website
tool = ScrapeWebsiteTool(website_url='https://www.example.com')
```

Arguments¶
website_url : Mandatory website URL to read the file. This is the primary input for the tool, specifying which website's content should be scraped and read.

DirectoryReadTool¶
Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
The DirectoryReadTool is a highly efficient utility designed for the comprehensive listing of directory contents. It recursively navigates through the specified directory, providing users with a detailed enumeration of all files, including those nested within subdirectories. This tool is

indispensable for tasks requiring a thorough inventory of directory structures or for validating the organization of files within directories.

Installation¶
Install the crewai_tools package to use the DirectoryReadTool in your project. If you haven't added this package to your environment, you can easily install it with pip using the following command:

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

This installs the latest version of the crewai_tools package, allowing access to the DirectoryReadTool and other utilities.

Example¶
The DirectoryReadTool is simple to use. The code snippet below shows how to set up and use the tool to list the contents of a specified directory:

```
from crewai_tools import DirectoryReadTool

# Initialize the tool so the agent can read any directory's content it learns about during execution
tool = DirectoryReadTool()

# OR

# Initialize the tool with a specific directory, so the agent can only read the content of the specified directory
tool = DirectoryReadTool(directory='/path/to/your/directory')
```

Arguments¶
The DirectoryReadTool requires minimal configuration for use. The essential argument for this tool is as follows:

directory: Optional A argument that specifies the path to the directory whose contents you wish to list. It accepts both absolute and relative paths, guiding the tool to the desired directory for content listing.

FileReadTool¶
Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶

The FileReadTool is a versatile component of the crewai_tools package, designed to streamline the process of reading and retrieving content from files. It is particularly useful in scenarios such as batch text file processing, runtime configuration file reading, and data importation for analytics. This tool supports various text-based file formats including .txt, .csv, .json and more, and adapts its functionality based on the file type, for instance, converting JSON content into a Python dictionary for easy use.

Installation¶
Install the crewai_tools package to use the FileReadTool in your projects:

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

Example¶
To get started with the FileReadTool:

```
from crewai_tools import FileReadTool

# Initialize the tool to read any files the agents knows or lean the path for
file_read_tool = FileReadTool()

# OR

# Initialize the tool with a specific file path, so the agent can only read the content of the specified file
file_read_tool = FileReadTool(file_path='path/to/your/file.txt')
```

Arguments¶
file_path: The path to the file you want to read. It accepts both absolute and relative paths. Ensure the file exists and you have the necessary permissions to access it.

SeleniumScrapingTool¶
Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is designed for efficient web scraping, enabling users to extract content from web pages. It supports targeted scraping by allowing the specification of a CSS selector for desired elements. The flexibility of the tool enables it to be used on any website URL provided by the user, making it a versatile tool for various web scraping needs.

Installation¶
Install the crewai_tools package

pip install 'crewai[tools]'
Example¶

```
from crewai_tools import SeleniumScrapingTool

# Example 1: Scrape any website it finds during its execution
tool = SeleniumScrapingTool()

# Example 2: Scrape the entire webpage
tool = SeleniumScrapingTool(website_url='https://example.com')

# Example 3: Scrape a specific CSS element from the webpage
tool = SeleniumScrapingTool(website_url='https://example.com', css_element='.main-content')

# Example 4: Scrape using optional parameters for customized scraping
tool = SeleniumScrapingTool(website_url='https://example.com', css_element='.main-content',
cookie={'name': 'user', 'value': 'John Doe'})
```

Arguments¶
website_url: Mandatory. The URL of the website to scrape.
css_element: Mandatory. The CSS selector for a specific element to scrape from the website.
cookie: Optional. A dictionary containing cookie information. This parameter allows the tool to simulate a session with cookie information, providing access to content that may be restricted to logged-in users.
wait_time: Optional. The number of seconds the tool waits after loading the website and after setting a cookie, before scraping the content. This allows for dynamic content to load properly.

DirectorySearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is designed to perform a semantic search for queries within the content of a specified directory. Utilizing the RAG (Retrieval-Augmented Generation) methodology, it offers a powerful means to semantically navigate through the files of a given directory. The tool can be

dynamically set to search any directory specified at runtime or can be pre-configured to search within a specific directory upon initialization.

Installation¶
To start using the DirectorySearchTool, you need to install the crewai_tools package. Execute the following command in your terminal:

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

Example¶
The following examples demonstrate how to initialize the DirectorySearchTool for different use cases and how to perform a search:

```
from crewai_tools import DirectorySearchTool

# To enable searching within any specified directory at runtime
tool = DirectorySearchTool()

# Alternatively, to restrict searches to a specific directory
tool = DirectorySearchTool(directory='/path/to/directory')
```

Arguments¶
directory : This string argument specifies the directory within which to search. It is mandatory if the tool has not been initialized with a directory; otherwise, the tool will only search within the initialized directory.

PDFSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
The PDFSearchTool is a RAG tool designed for semantic searches within PDF content. It allows for inputting a search query and a PDF document, leveraging advanced search techniques to find relevant content efficiently. This capability makes it especially useful for extracting specific information from large PDF files quickly.

Installation¶
To get started with the PDFSearchTool, first, ensure the crewai_tools package is installed with the following command:

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

Example¶
Here's how to use the PDFSearchTool to search within a PDF document:

```
from crewai_tools import PDFSearchTool

# Initialize the tool allowing for any PDF content search if the path is provided during execution
tool = PDFSearchTool()

# OR

# Initialize the tool with a specific PDF path for exclusive search within that document
tool = PDFSearchTool(pdf='path/to/your/document.pdf')
```

Arguments¶
pdf: Optinal The PDF path for the search. Can be provided at initialization or within the run method's arguments. If provided at initialization, the tool confines its search to the specified document.

TXTSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is used to perform a RAG (Retrieval-Augmented Generation) search within the content of a text file. It allows for semantic searching of a query within a specified text file's content, making it an invaluable resource for quickly extracting information or finding specific sections of text based on the query provided.

Installation¶

To use the TXTSearchTool, you first need to install the crewai_tools package. This can be done using pip, a package manager for Python. Open your terminal or command prompt and enter the following command:

pip install 'crewai[tools]'
This command will download and install the TXTSearchTool along with any necessary dependencies.

Example¶
The following example demonstrates how to use the TXTSearchTool to search within a text file. This example shows both the initialization of the tool with a specific text file and the subsequent search within that file's content.

```
from crewai_tools import TXTSearchTool

# Initialize the tool to search within any text file's content the agent learns about during its execution
tool = TXTSearchTool()

# OR

# Initialize the tool with a specific text file, so the agent can search within the given text file's content
tool = TXTSearchTool(txt='path/to/text/file.txt')
```

Arguments¶
txt (str): Optinal. The path to the text file you want to search. This argument is only required if the tool was not initialized with a specific text file; otherwise, the search will be conducted within the initially provided text file.

CSVSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶

This tool is used to perform a RAG (Retrieval-Augmented Generation) search within a CSV file's content. It allows users to semantically search for queries in the content of a specified CSV file. This feature is particularly useful for extracting information from large CSV datasets where traditional search methods might be inefficient. All tools with "Search" in their name, including CSVSearchTool, are RAG tools designed for searching different sources of data.

Installation¶
Install the crewai_tools package

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

Example¶

```
from crewai_tools import CSVSearchTool

# Initialize the tool with a specific CSV file. This setup allows the agent to only search the given CSV file.
tool = CSVSearchTool(csv='path/to/your/csvfile.csv')

# OR

# Initialize the tool without a specific CSV file. Agent  will need to provide the CSV path at runtime.
tool = CSVSearchTool()
```

Arguments¶
csv : The path to the CSV file you want to search. This is a mandatory argument if the tool was initialized without a specific CSV file; otherwise, it is optional.

JSONSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is used to perform a RAG search within a JSON file's content. It allows users to initiate a search with a specific JSON path, focusing the search operation within that particular JSON

file. If the path is provided at initialization, the tool restricts its search scope to the specified JSON file, thereby enhancing the precision of search results.

Installation¶
Install the crewai_tools package by executing the following command in your terminal:

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

Example¶
Below are examples demonstrating how to use the JSONSearchTool for searching within JSON files. You can either search any JSON content or restrict the search to a specific JSON file.

```
from crewai_tools import JSONSearchTool

# Example 1: Initialize the tool for a general search across any JSON content. This is useful when the path is known or can be discovered during execution.
tool = JSONSearchTool()

# Example 2: Initialize the tool with a specific JSON path, limiting the search to a particular JSON file.
tool = JSONSearchTool(json_path='./path/to/your/file.json')
```

Arguments¶
json_path (str): An optional argument that defines the path to the JSON file to be searched. This parameter is only necessary if the tool is initialized without a specific JSON path. Providing this argument restricts the search to the specified JSON file.

WebsiteSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is specifically crafted for conducting semantic searches within the content of a particular website. Leveraging a Retrieval-Augmented Generation (RAG) model, it navigates through the information provided on a given URL. Users have the flexibility to either initiate a

search across any website known or discovered during its usage or to concentrate the search on a predefined, specific website.

Installation¶
Install the crewai_tools package by executing the following command in your terminal:

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

Example¶
To utilize the WebsiteSearchTool for different use cases, follow these examples:

```
from crewai_tools import WebsiteSearchTool

# To enable the tool to search any website the agent comes across or learns about during its operation
tool = WebsiteSearchTool()

# OR

# To restrict the tool to only search within the content of a specific website.
tool = WebsiteSearchTool(website='https://example.com')
```

Arguments¶
website : An optional argument that specifies the valid website URL to perform the search on. This becomes necessary if the tool is initialized without a specific website. In the WebsiteSearchToolSchema, this argument is mandatory. However, in the FixedWebsiteSearchToolSchema, it becomes optional if a website is provided during the tool's initialization, as it will then only search within the predefined website's content.

GitHubSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
The GitHubSearchTool is a Read, Append, and Generate (RAG) tool specifically designed for conducting semantic searches within GitHub repositories. Utilizing advanced semantic search

capabilities, it sifts through code, pull requests, issues, and repositories, making it an essential tool for developers, researchers, or anyone in need of precise information from GitHub.

Installation¶
To use the GitHubSearchTool, first ensure the crewai_tools package is installed in your Python environment:

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

This command installs the necessary package to run the GitHubSearchTool along with any other tools included in the crewai_tools package.

Example¶
Here's how you can use the GitHubSearchTool to perform semantic searches within a GitHub repository:

```
from crewai_tools import GitHubSearchTool

# Initialize the tool for semantic searches within a specific GitHub repository
tool = GitHubSearchTool(
    github_repo='https://github.com/example/repo',
    content_types=['code', 'issue'] # Options: code, repo, pr, issue
)

# OR

# Initialize the tool for semantic searches within a specific GitHub repository, so the agent can search any repository if it learns about during its execution
tool = GitHubSearchTool(
    content_types=['code', 'issue'] # Options: code, repo, pr, issue
)
```

Arguments¶
github_repo : The URL of the GitHub repository where the search will be conducted. This is a mandatory field and specifies the target repository for your search.
content_types : Specifies the types of content to include in your search. You must provide a list of content types from the following options: code for searching within the code, repo for searching within the repository's general information, pr for searching within pull requests, and issue for searching within issues. This field is mandatory and allows tailoring the search to specific content types within the GitHub repository.

YoutubeVideoSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is part of the crewai_tools package and is designed to perform semantic searches within Youtube video content, utilizing Retrieval-Augmented Generation (RAG) techniques. It is one of several "Search" tools in the package that leverage RAG for different sources. The YoutubeVideoSearchTool allows for flexibility in searches; users can search across any Youtube video content without specifying a video URL, or they can target their search to a specific Youtube video by providing its URL.

Installation¶
To utilize the YoutubeVideoSearchTool, you must first install the crewai_tools package. This package contains the YoutubeVideoSearchTool among other utilities designed to enhance your data analysis and processing tasks. Install the package by executing the following command in your terminal:

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

Example¶
To integrate the YoutubeVideoSearchTool into your Python projects, follow the example below. This demonstrates how to use the tool both for general Youtube content searches and for targeted searches within a specific video's content.

```
from crewai_tools import YoutubeVideoSearchTool

# General search across Youtube content without specifying a video URL, so the agent can search within any Youtube video content it learns about irs url during its operation
tool = YoutubeVideoSearchTool()

# Targeted search within a specific Youtube video's content
tool = YoutubeVideoSearchTool(youtube_video_url='https://youtube.com/watch?v=example')
```

Arguments¶
The YoutubeVideoSearchTool accepts the following initialization arguments:

youtube_video_url: An optional argument at initialization but required if targeting a specific Youtube video. It specifies the Youtube video URL path you want to search within.

YoutubeChannelSearchTool¶
Depend on OpenAI

All RAG tools at the moment can only use openAI to generate embeddings, we are working on adding support for other providers.

Experimental

We are still working on improving tools, so there might be unexpected behavior or changes in the future.

Description¶
This tool is designed to perform semantic searches within a specific Youtube channel's content. Leveraging the RAG (Retrieval-Augmented Generation) methodology, it provides relevant search results, making it invaluable for extracting information or finding specific content without the need to manually sift through videos. It streamlines the search process within Youtube channels, catering to researchers, content creators, and viewers seeking specific information or topics.

Installation¶
To utilize the YoutubeChannelSearchTool, the crewai_tools package must be installed. Execute the following command in your shell to install:

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

Example¶
To begin using the YoutubeChannelSearchTool, follow the example below. This demonstrates initializing the tool with a specific Youtube channel handle and conducting a search within that channel's content.

```
from crewai_tools import YoutubeChannelSearchTool

# Initialize the tool to search within any Youtube channel's content the agent learns about during its execution
tool = YoutubeChannelSearchTool()

# OR

# Initialize the tool with a specific Youtube channel handle to target your search
tool = YoutubeChannelSearchTool(youtube_channel_handle='@exampleChannel')
```

Arguments¶

youtube_channel_handle : A mandatory string representing the Youtube channel handle. This parameter is crucial for initializing the tool to specify the channel you want to search within. The tool is designed to only search within the content of the provided channel handle.

Examples

AI Crew for Instagram Post
Introduction
This project is an example using the CrewAI framework to automate the process of coming up with an instagram post. CrewAI orchestrates autonomous AI agents, enabling them to collaborate and execute complex tasks efficiently.

Instagram Post
Instagram Post

By @joaomdmoura

CrewAI Framework
Running the script
Details & Explanation
Using Local Models with Ollama
License
CrewAI Framework
CrewAI is designed to facilitate the collaboration of role-playing AI agents. In this example, these agents work together to give a complete stock analysis and investment recommendation

Running the Script
This example uses OpenHermes 2.5 through Ollama by default so you should to download Ollama and OpenHermes.

You can change the model by changing the MODEL env var in the .env file.

Configure Environment: Copy ``.env.example` and set up the environment variables for Browseless, Serper.
Install Dependencies: Run poetry install --no-root.
Execute the Script: Run python main.py and input your idea.
Details & Explanation
Running the Script: Execute `python main.py`` and input your idea when prompted. The script will leverage the CrewAI framework to process the idea and generate a landing page.
Key Components:
./main.py: Main script file.

./tasks.py: Main file with the tasks prompts.
./agents.py: Main file with the agents creation.
./tools/: Contains tool classes used by the agents.

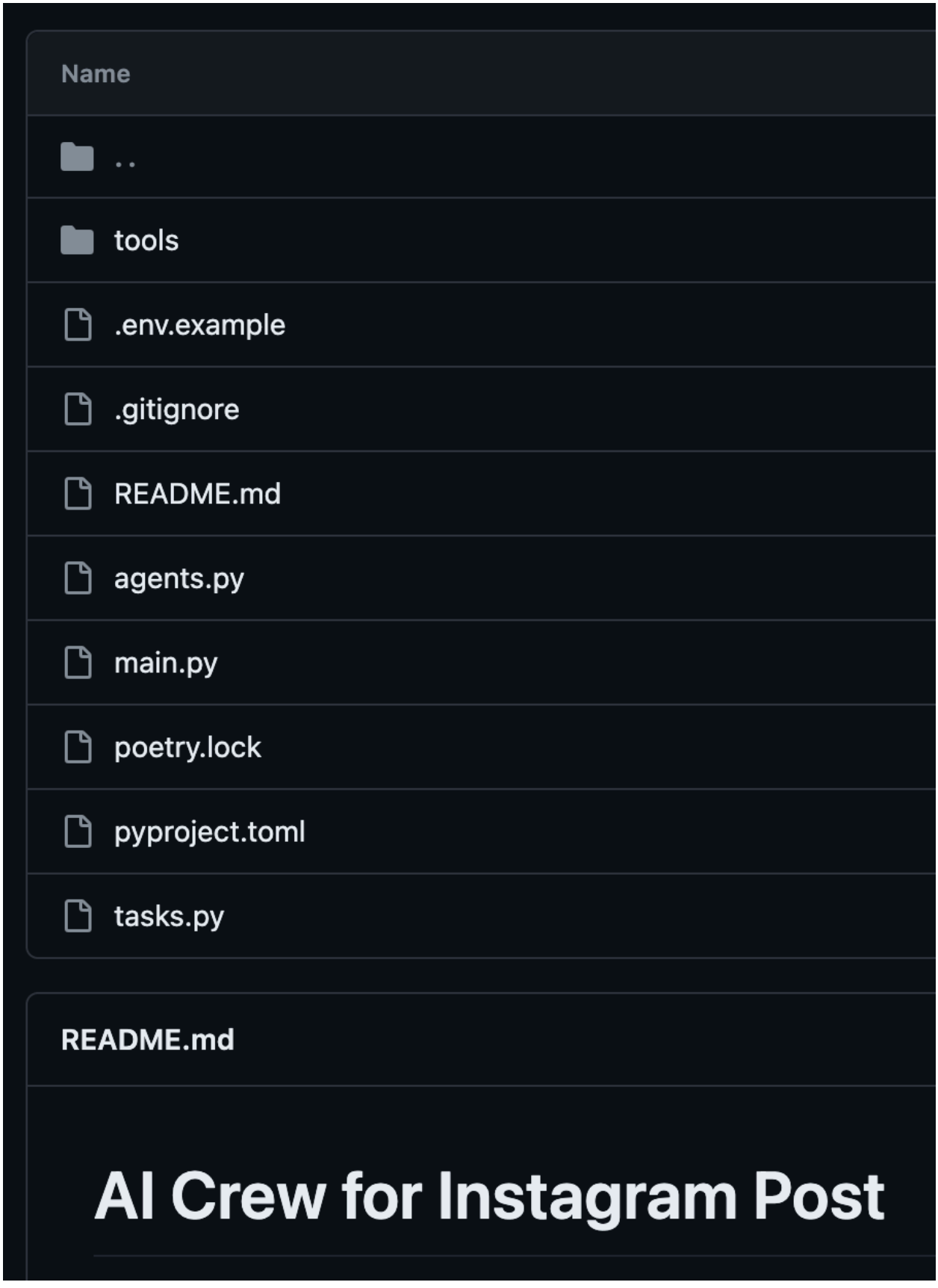
Using Local Models with Ollama

This example run enterily local models, the CrewAI framework supports integration with both closed and local models, by using tools such as Ollama, for enhanced flexibility and customization. This allows you to utilize your own models, which can be particularly useful for specialized tasks or data privacy concerns.

Setting Up Ollama

Install Ollama: Ensure that Ollama is properly installed in your environment. Follow the installation guide provided by Ollama for detailed instructions.
Configure Ollama: Set up Ollama to work with your local model. You will probably need to tweak the model using a Modelfile, I'd recommend playing with top_p and temperature.

License


Name
..
tools
.env.example
.gitignore
README.md
agents.py
main.py
poetry.lock
pyproject.toml
tasks.py
README.md
AI Crew for Instagram Post

```
 # main.py

from dotenv import load_dotenv
load_dotenv()

from textwrap import dedent
from crewai import Agent, Crew

from tasks import MarketingAnalysisTasks
from agents import MarketingAnalysisAgents

tasks = MarketingAnalysisTasks()
agents = MarketingAnalysisAgents()

print("## Welcome to the marketing Crew")
print('-------------------------------')
product_website = input("What is the product website you want a marketing strategy for?\n")
product_details = input("Any extra details about the product and or the instagram post you
want?\n")


# Create Agents
product_competitor_agent = agents.product_competitor_agent()
strategy_planner_agent = agents.strategy_planner_agent()
creative_agent = agents.creative_content_creator_agent()
# Create Tasks
website_analysis = tasks.product_analysis(product_competitor_agent, product_website,
product_details)
market_analysis = tasks.competitor_analysis(product_competitor_agent, product_website,
product_details)
campaign_development = tasks.campaign_development(strategy_planner_agent,
product_website, product_details)
write_copy = tasks.instagram_ad_copy(creative_agent)

# Create Crew responsible for Copy
copy_crew = Crew(
        agents=[
                product_competitor_agent,
                strategy_planner_agent,
                creative_agent
        ],
        tasks=[
                website_analysis,
                market_analysis,
```

```
                campaign_development,
                write_copy
        ],
        verbose=True
)

ad_copy = copy_crew.kickoff()

# Create Crew responsible for Image
senior_photographer = agents.senior_photographer_agent()
chief_creative_diretor = agents.chief_creative_diretor_agent()
# Create Tasks for Image
take_photo = tasks.take_photograph_task(senior_photographer, ad_copy, product_website,
product_details)
approve_photo = tasks.review_photo(chief_creative_diretor, product_website, product_details)

image_crew = Crew(
        agents=[
                senior_photographer,
                chief_creative_diretor
        ],
        tasks=[
                take_photo,
                approve_photo
        ],
        verbose=True
)

image = image_crew.kickoff()

# Print results
print("\n\n############################")
print("## Here is the result")
print("############################\n")
print("Your post copy:")
print(ad_copy)
print("'\n\nYour midjourney description:")
print(image)


# agents.py

import os
from textwrap import dedent
```

```
from crewai import Agent
from tools.browser_tools import BrowserTools
from tools.search_tools import SearchTools
from langchain.agents import load_tools

from langchain.llms import Ollama

class MarketingAnalysisAgents:
	def __init__(self):
		self.llm = Ollama(model=os.environ['MODEL'])

	def product_competitor_agent(self):
		return Agent(
			role="Lead Market Analyst",
			goal=dedent("""\
				Conduct amazing analysis of the products and
				competitors, providing in-depth insights to guide
				marketing strategies."""),
			backstory=dedent("""\
				As the Lead Market Analyst at a premier
				digital marketing firm, you specialize in dissecting
				online business landscapes."""),
			tools=[
					BrowserTools.scrape_and_summarize_website,
					SearchTools.search_internet
			],
			allow_delegation=False,
			llm=self.llm,
			verbose=True
		)

	def strategy_planner_agent(self):
		return Agent(
			role="Chief Marketing Strategist",
			goal=dedent("""\
				Synthesize amazing insights from product analysis
				to formulate incredible marketing strategies."""),
			backstory=dedent("""\
				You are the Chief Marketing Strategist at
				a leading digital marketing agency, known for crafting
				bespoke strategies that drive success."""),
			tools=[
					BrowserTools.scrape_and_summarize_website,
					SearchTools.search_internet,
```

```
                SearchTools.search_instagram
            ],
            llm=self.llm,
            verbose=True
        )

    def creative_content_creator_agent(self):
        return Agent(
            role="Creative Content Creator",
            goal=dedent("""\
                Develop compelling and innovative content
                for social media campaigns, with a focus on creating
                high-impact Instagram ad copies."""),
            backstory=dedent("""\
                As a Creative Content Creator at a top-tier
                digital marketing agency, you excel in crafting narratives
                that resonate with audiences on social media.
                Your expertise lies in turning marketing strategies
                into engaging stories and visual content that capture
                attention and inspire action."""),
            tools=[
                BrowserTools.scrape_and_summarize_website,
                SearchTools.search_internet,
                SearchTools.search_instagram
            ],
            llm=self.llm,
            verbose=True
        )

    def senior_photographer_agent(self):
        return Agent(
            role="Senior Photographer",
            goal=dedent("""\
                Take the most amazing photographs for instagram ads that
                capture emotions and convey a compelling message."""),
            backstory=dedent("""\
                As a Senior Photographer at a leading digital marketing
                agency, you are an expert at taking amazing photographs
that
                inspire and engage, you're now working on a new
campaign for a super
                important customer and you need to take the most
amazing photograph."""),
            tools=[
```

```
                BrowserTools.scrape_and_summarize_website,
                SearchTools.search_internet,
                SearchTools.search_instagram
            ],
            llm=self.llm,
            allow_delegation=False,
            verbose=True
        )

    def chief_creative_diretor_agent(self):
        return Agent(
            role="Chief Creative Director",
            goal=dedent("""\
                Oversee the work done by your team to make sure it's the
best
                possible and aligned with the product's goals, review,
approve,
                ask clarifying question or delegate follow up work if
necessary to make
                decisions"""),
            backstory=dedent("""\
                You're the Chief Content Officer of leading digital
                marketing specialized in product branding. You're working
on a new
                customer, trying to make sure your team is crafting the
best possible
                content for the customer."""),
            tools=[
                BrowserTools.scrape_and_summarize_website,
                SearchTools.search_internet,
                SearchTools.search_instagram
            ],
            llm=self.llm,
            verbose=True
        )


# tasks.py

from crewai import Task
from textwrap import dedent

class MarketingAnalysisTasks:
    def product_analysis(self, agent, product_website, product_details):
```

```
        return Task(description=dedent(f"""\
            Analyze the given product website: {product_website}.
            Extra details provided by the customer: {product_details}.

            Focus on identifying unique features, benefits,
            and the overall narrative presented.

            Your final report should clearly articulate the
            product's key selling points, its market appeal,
            and suggestions for enhancement or positioning.
            Emphasize the aspects that make the product stand out.

            Keep in mind, attention to detail is crucial for
            a comprehensive analysis. It's currenlty 2024.
            """),
            agent=agent
        )

    def competitor_analysis(self, agent, product_website, product_details):
        return Task(description=dedent(f"""\
            Explore competitor of: {product_website}.
            Extra details provided by the customer: {product_details}.

            Identify the top 3 competitors and analyze their
            strategies, market positioning, and customer perception.

            Your final report MUST include BOTH all context about {product_website}
            and a detailed comparison to whatever competitor they have competitors.
            """),
            agent=agent
        )

    def campaign_development(self, agent, product_website, product_details):
        return Task(description=dedent(f"""\
            You're creating a targeted marketing campaign for: {product_website}.
            Extra details provided by the customer: {product_details}.

            To start this campaing we will need a strategy and creative content ideas.
            It should be meticulously designed to captivate and engage
            the product's target audience.

            Based on your ideas your co-workers will create the content for the
campaign.
```

```
                Your final answer MUST be ideas that will resonate with the audience and
                also include ALL context you have about the product and the customer.
                """),
                agent=agent
            )

    def instagram_ad_copy(self, agent):
        return Task(description=dedent("""\
                Craft an engaging Instagram post copy.
                The copy should be punchy, captivating, concise,
                and aligned with the product marketing strategy.

                Focus on creating a message that resonates with
                the target audience and highlights the product's
                unique selling points.

                Your ad copy must be attention-grabbing and should
                encourage viewers to take action, whether it's
                visiting the website, making a purchase, or learning
                more about the product.

                Your final answer MUST be 3 options for an ad copy for instagram that
                not only informs but also excites and persuades the audience.
                """),
                agent=agent
            )

    def take_photograph_task(self, agent, copy, product_website, product_details):
        return Task(description=dedent(f"""\
                You are working on a new campaign for a super important customer,
                and you MUST take the most amazing photo ever for an instagram post
                regarding the product, you have the following copy:
                {copy}

                This is the product you are working with: {product_website}.
                Extra details provided by the customer: {product_details}.

                Imagine what the photo you wanna take describe it in a paragraph.
                Here are some examples for you follow:
                - high tech airplaine in a beautiful blue sky in a beautiful sunset super
cripsy beautiful 4k, professional wide shot
                - the last supper, with Jesus and his disciples, breaking bread, close shot,
soft lighting, 4k, crisp
```

```
                - an bearded old man in the snows, using very warm clothing, with
mountains full of snow behind him, soft lighting, 4k, crisp, close up to the camera

                Think creatively and focus on how the image can capture the audience's
                attention. Don't show the actual product on the photo.

                Your final answer must be 3 options of photographs, each with 1
paragraph
                describing the photograph exactly like the examples provided above.
                """),
                agent=agent
            )

    def review_photo(self, agent, product_website, product_details):
        return Task(description=dedent(f"""\
                Review the photos you got from the senior photographer.
                Make sure it's the best possible and aligned with the product's goals,
                review, approve, ask clarifying question or delegate follow up work if
                necessary to make decisions. When delegating work send the full draft
                as part of the information.

                This is the product you are working with: {product_website}.
                Extra details provided by the customer: {product_details}.

                Here are some examples on how the final photographs should look like:
                - high tech airplaine in a beautiful blue sky in a beautiful sunset super
cripsy beautiful 4k, professional wide shot
                - the last supper, with Jesus and his disciples, breaking bread, close shot,
soft lighting, 4k, crisp
                - an bearded old man in the snows, using very warm clothing, with
mountains full of snow behind him, soft lighting, 4k, crisp, close up to the camera

                Your final answer must be 3 reviewed options of photographs,
                each with 1 paragraph description following the examples provided
above.
                """),
                agent=agent
            )
```