Mar 9, 2025

Advanced Techniques to Build Robust Agentic Systems (Part A)

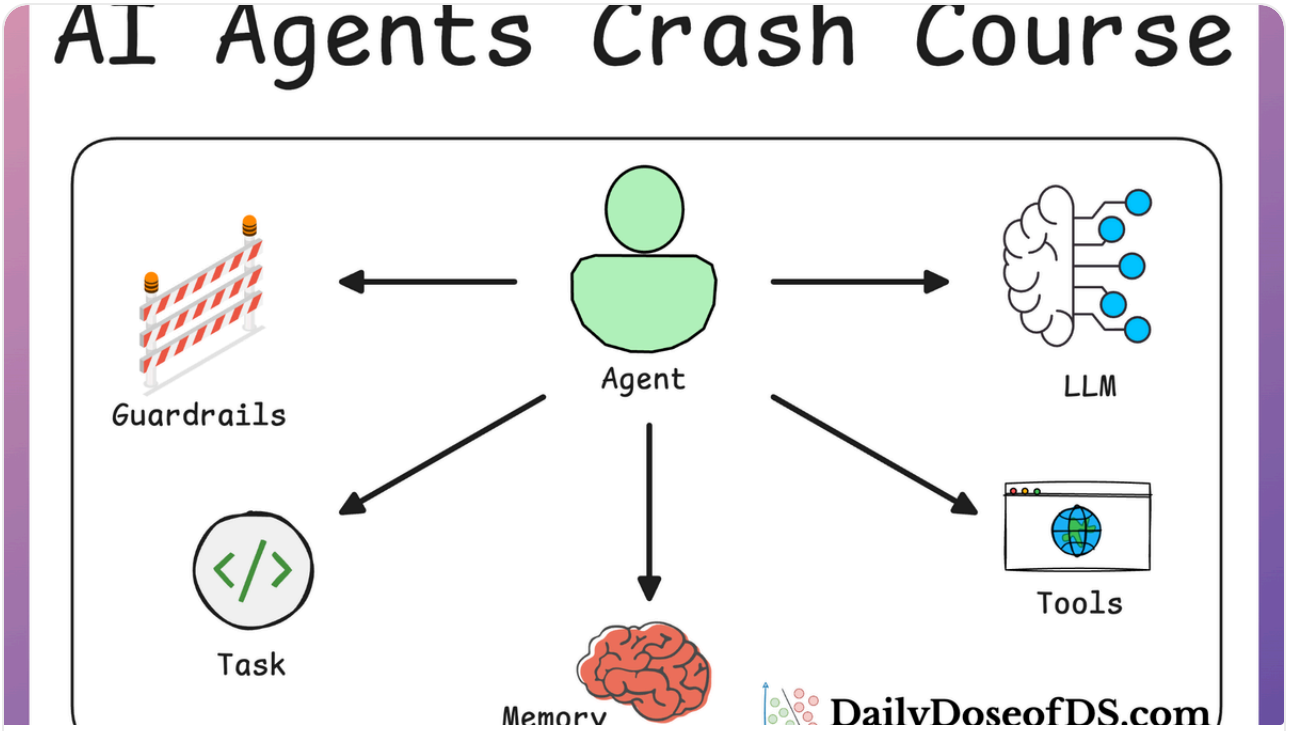
AI Agents Crash Course—Part 5 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

Over the past four parts of this crash course, we have progressively built our understanding of Agentic Systems and multi-agent workflows with CrewAI.

undamentals of Agentic systems, understanding how AI agents can act autonomously to perform structured tasks.

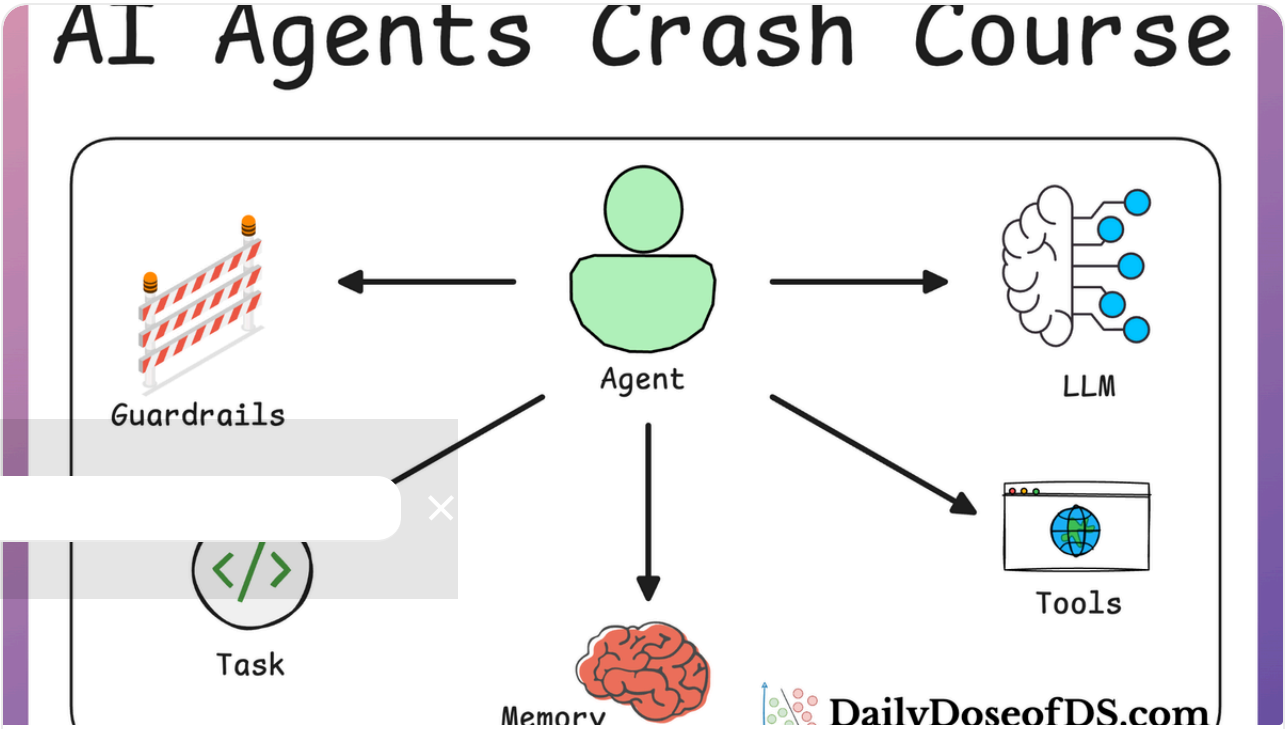


AI Agents Crash Course—Part 1 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 2, we explored how to extend Agent capabilities by integrating custom tools, using structured tools, and building modular Crews to compartmentalize responsibilities.

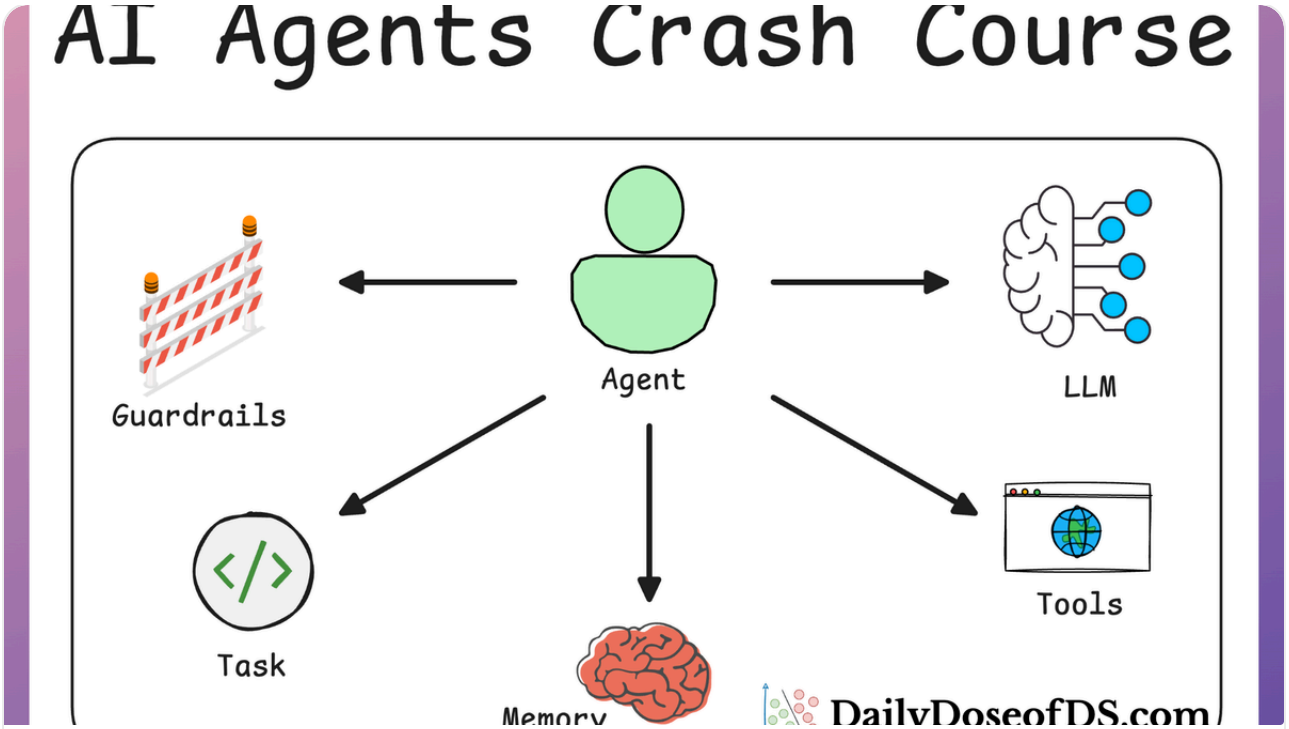


AI Agents Crash Course—Part 2 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 3, we focused on Flows, learning about state management, flow control, and integrating a Crew into a Flow. As discussed last time, with Flows, you can create structured, event-driven workflows that seamlessly connect multiple tasks, manage state, and control the flow of execution in your AI applications.

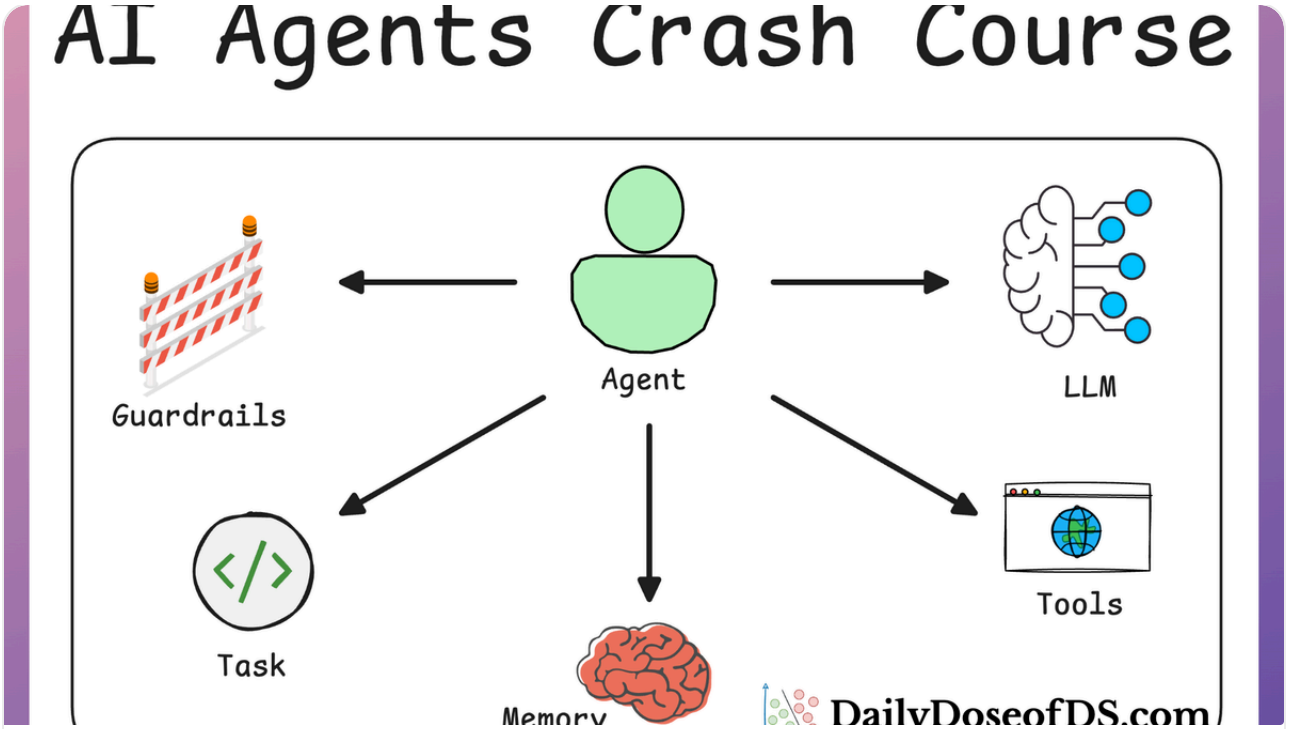


AI Agents Crash Course—Part 3 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 4, we extended these concepts into real-world multi-agent, multi-crew Flow projects, demonstrating how to automate complex workflows such as content planning and book writing.



AI Agents Crash Course—Part 4 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

Now, in Part 5 and 6, we shall move into advanced techniques that make AI agents more robust, dynamic, and adaptable.

- Guardrails → Enforcing constraints to ensure agents produce reliable and expected outputs.
- Referencing other Tasks and their outputs → Allowing agents to dynamically use previous task results.
- Executing tasks async → Running agent tasks concurrently to optimize performance.
- Adding callbacks → Allowing post-processing or monitoring of task completions.
- Introduce human-in-the-loop during execution → Introducing human-in-the-loop mechanisms for validation and control.
- Hierarchical Agentic processes → Structuring agents into sub-agents and multi-level execution trees for more complex workflows.

- Multimodal Agents → Extending CrewAI agents to handle text, images, audio, and beyond.
- and more.

We have split this into two parts to improve the reading experience and ensure that you do not feel overwhelmed with several details.

Let's dive in!

> 💡 If you haven't read Part 1 to Part 4 yet, we highly recommend doing so before moving ahead.

Installation and setup

> 💡 Feel free to skip this part if you have followed these instructions before.

Throughout this crash course, we have been using CrewAI, an open-source framework that makes it seamless to orchestrate role-playing, set goals, integrate tools, bring any of the popular LLMs, etc., to build autonomous AI agents.

GitHub - crewAIInc/crewAI: Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks.

Framework for orchestrating role-playing, autonomous AI agents. By fostering ...CrewAI empowers agents to work together seamlessly, tackling...

GitHub • crewAIInc

To highlight more, CrewAI is a standalone independent framework without any dependencies on Langchain or other agent frameworks.

Let's dive in!

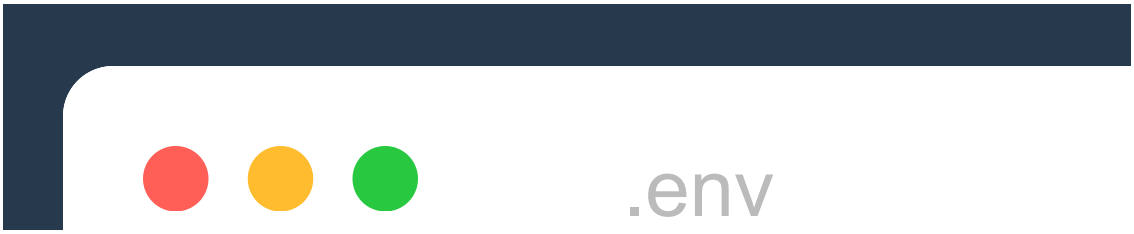
To get started, install CrewAI as follows:

Like the RAG crash course, we shall be using Ollama to serve LLMs locally. That said, CrewAI integrates with several LLM providers like:

- OpenAI
- Gemini
- Groq
- Azure
- Fireworks AI
- Cerebras
- SambaNova
- and many more.

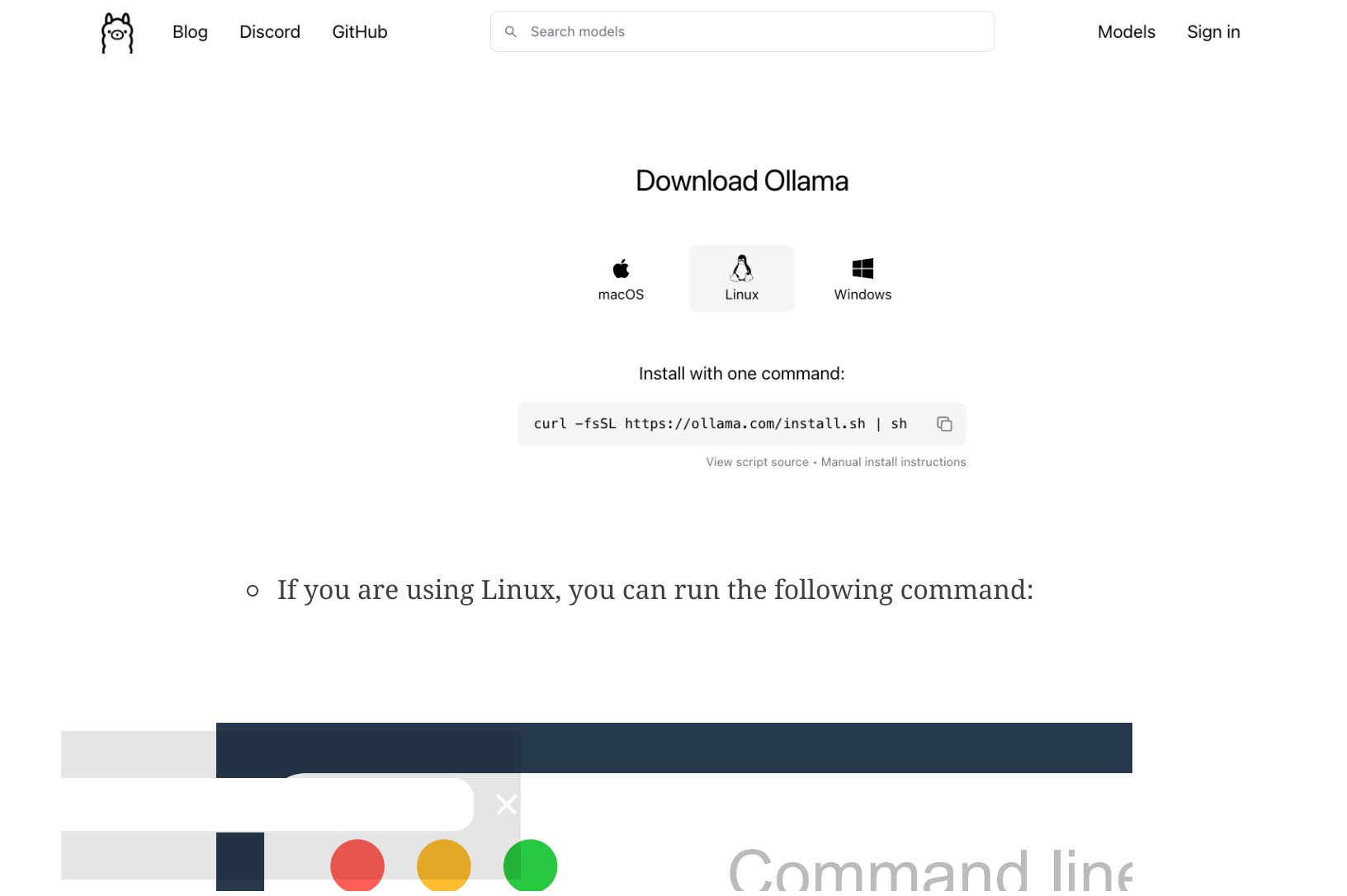
> 💡 If you have an OpenAI API key, we recommend using that since the outputs may not make sense at times with weak LLMs. If you don't have an API key, you can get free credits by creating a dummy account on OpenAI and use that instead. If not, you can continue reading and use Ollama instead but the outputs could be poor in that case.

To set up OpenAI, create a `.env` file in the current directory and specify your OpenAI API key as follows:



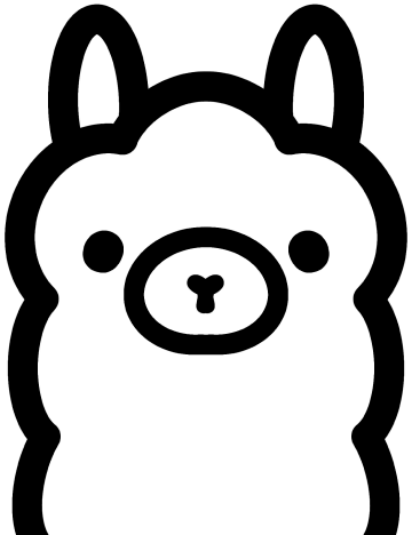
Also, here's a step-by-step guide on using Ollama:

- Go to **Ollama.com**, select your operating system, and follow the instructions.



- If you are using Linux, you can run the following command:

- Ollama supports a bunch of models that are also listed in the model library:



library

Get up and running with large language models.

Blog Discord GitHub Search models Models Sign in Download

Models

Filter by name... Most popular

llama3.2

Meta's Llama 3.2 goes small with 1B and 3B models.

tools 1b 3b

2.2M Pulls 63 Tags Updated 5 weeks ago

llama3.1

Llama 3.1 is a new state-of-the-art model from Meta available in 8B, 70B and 405B parameter sizes.

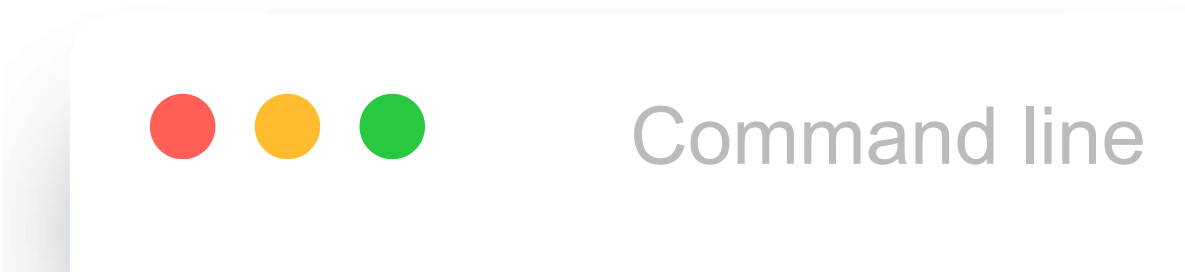
tools 8b 70b 405b

8M Pulls 93 Tags Updated 7 weeks ago

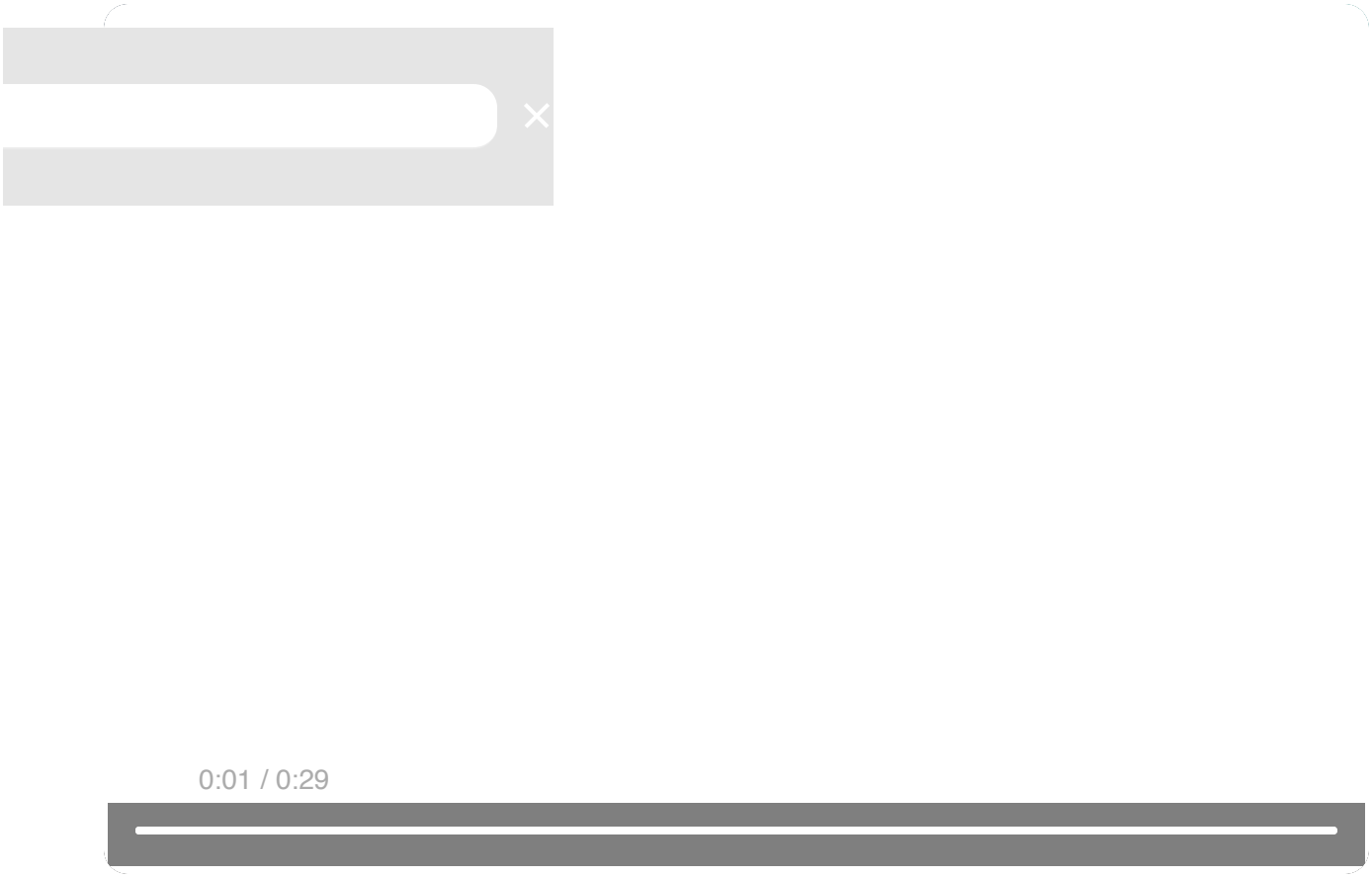
gemma2

Google Gemma 2 is a high-performing and efficient model available in

Once you've found the model you're looking for, run this command in your terminal:



The above command will download the model locally, so give it some time to complete. But once it's done, you'll have Llama 3.2 3B running locally, as shown below, which depicts Microsoft's Phi-3 served locally through Ollama:



That said, for our demo, we would be running Llama 3.2 1B model instead since it's smaller and will not take much memory:

Done!

Everything is set up now and we can move on to building robust agentic workflows.

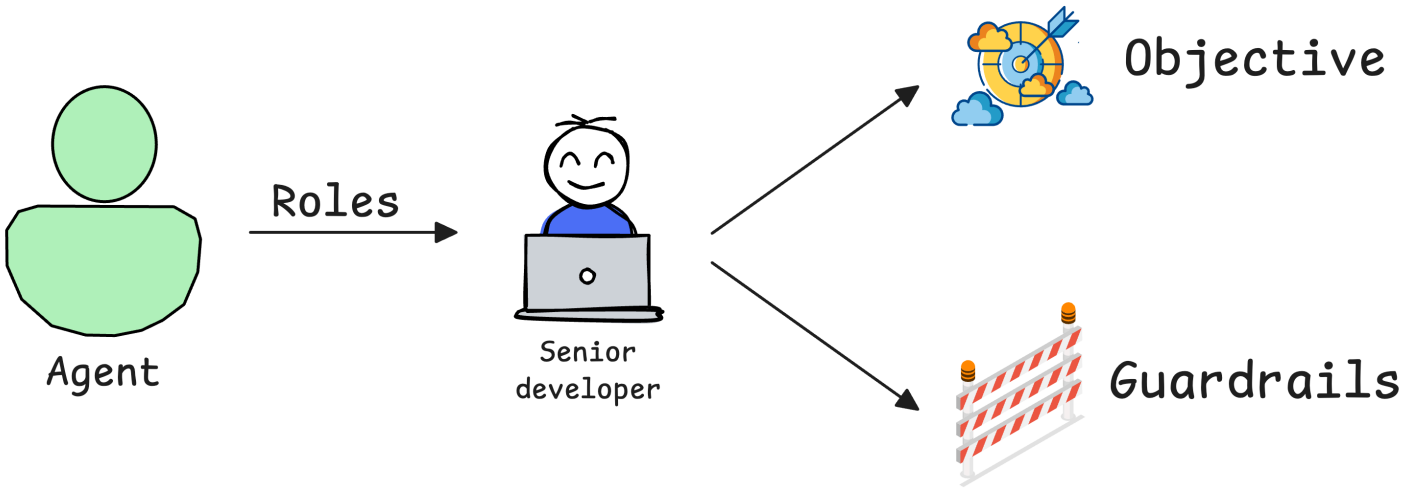
You can download the code for this article below:

notebook.ipynb • 28 KB

Guardrails

AI agents are powerful, but without constraints and safeguards, they can hallucinate, enter infinite loops, or make unreliable decisions.

Guardrails ensure that agents stay on track and maintain quality standards.



Examples of useful guardrails include:

- Limiting tool usage: Prevent an agent from overusing APIs or generating irrelevant queries.
- Setting validation checkpoints: Ensure outputs meet predefined criteria before moving to the next step.
- Establishing fallback mechanisms: If an agent fails to complete a task, another agent or human reviewer can intervene.

For instance, in an AI-powered legal assistant, guardrails can prevent the system from:

- Generating non-factual legal advice.
- Misinterpreting laws that differ across jurisdictions
- Citing outdated or incorrect legal precedents.

Thus, if you can, you should implement safeguards, validation layers, and fallback mechanisms to keep agents aligned with expected behavior.

Let's understand the implementation below.

Let’s say we have an AI agent summarizing research papers. We want to ensure:

- The summary is under 150 words.

- The summary is not empty.

Here's how we add a guardrail to validate the output.

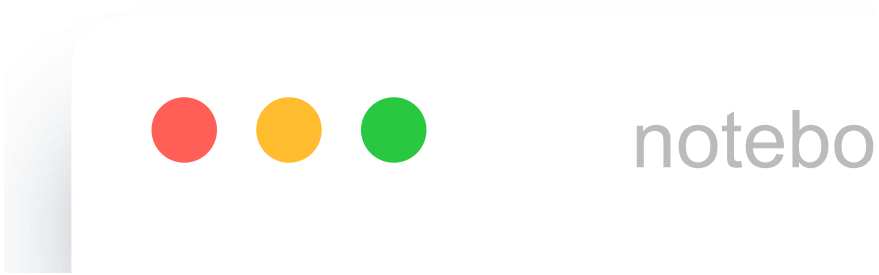
notebook.ipynb

Ensure AI-generated summaries stay under 150 words.

```python
def validate_summary_length(task_output):
    try:
        print("Validating summary length")
        task_str_output = str(task_output)
        total_words = len(task_str_output.split())

        print(f"Word count: {total_words}")

        if total_words > 150:
            print("Summary exceeds 150 words")
            return (False, f"""Summary exceeds 150 words.
                               Current Word count: {total_words}""")

        if total_words == 0:
            print("Summary is empty")
            return (False, "Generated summary is empty.")

        print("Summary is valid")
        return (True, task_output)

    except Exception as e:
        print("Validation system error")
        return (False, f"Validation system error: {str(e)}")
```

Let's break down the return structure of guardrail functions using the `validate_summary_length()` example above.

A guardrail function must return a tuple of two values:



- The first value (Boolean) → Indicates success (`True`) or failure (`False`).
- The second value → Provides either the validated result (if successful) or an error message as a string (if validation fails).

We have followed the same format in our guardrail method discussed above:

Finally, the guardrail function must accept just one parameter—which is the output of the Agent.

Let's use the above guardrail within an Agent and a Task to see this in action.

Below, we have defined a simple summarizer Agent and its Task:

notebook.ipynb

```
from crewai import Task, Agent

summary_agent = Agent(
    role="Summary Agent",
    goal="""Summarize the research paper
            'Convolutional Neural Networks' in 150 words.""",
    backstory="You specialize in summarizing research papers.",
    verbose=True
)

summary_task = Task(
    description="Summarize a research paper in 150 words.",
    expected_output="A concise research summary 150 words.",
    agent=summary_agent,
    guardrail=validate_summary_length,
)
```

Specify guardrail function

Next, we define a Crew and kick off as follows:

This produces the following output, and as we can see below, we notice a few print statements from the guardrail method—which raised no failure:

```
from crewai import Crew

        agents=[summary_agent],
        tasks=[summary_task],
        verbose=True
    )

    result = summary_crew.kickoff()
✓ 1.5s

Overriding of current TracerProvider is not allowed
# Agent: Summary Agent
## Task: Summarize a research paper in 150 words.


# Agent: Summary Agent
## Final Answer:
Convolutional Neural Networks (CNNs) are a specialized type of artificial neu


Validating summary length
Word count: 148
Summary is valid
```

But now let's understand what happens upon failure.

Every time a guardrail method returns this: `(False, error)`, the `error` (represented as a string) is sent to the Agent. The Agent then considers the error message to fix the failure cause.

This cycle repeats until the error is fixed, i.e, we get return value in the format: `(True, task_output)` OR, until a specified number of retries (3, by default) are reached.

Let's look at it below.

This time, let's intentionally specify in the fields about generating a 200 word summary instead while keeping the guardrail method same at 150 words:

We run this again as follows:

This time, we get the following output where the lines in yellow indicate the error message received from the guardrail method:

```
Overriding of current TracerProvider is not allowed
# Agent: Summary Agent
## Task: Summarize a research paper in 200 words.


The paper discusses advancements that have led to improvements in CNNs, such as the incorporation of activation functions like ReLU, normalization techniques, and various architectures includin


Validating summary length
Word count: 193
Summary exceeds 150 words
 Guardrail blocked, retrying, due to: Summary exceeds 150 words. Word count: 193

# Agent: Summary Agent
## Task: Summarize a research paper in 200 words.


# Agent: Summary Agent
## Final Answer:
The paper on Convolutional Neural Networks (CNNs) examines their structure and effectiveness in visual perception tasks. CNNs are engineered to learn spatial hierarchies of features directly fr

The paper outlines significant advancements in CNN architecture, including the implementation of activation functions like ReLU, normalization methods, and notable architectures, including Alex


Validating summary length
Word count: 168
Summary exceeds 150 words
 Guardrail blocked, retrying, due to: Summary exceeds 150 words. Word count: 168

# Agent: Summary Agent
## Task: Summarize a research paper in 200 words.


# Agent: Summary Agent
## Final Answer:
The research paper on Convolutional Neural Networks (CNNs) explores their architecture and applications in visual perception. CNNs are designed to automatically learn spatial hierarchies of fea


Validating summary length
Word count: 163
Summary exceeds 150 words
 Guardrail blocked, retrying, due to: Summary exceeds 150 words. Word count: 163

# Agent: Summary Agent
## Task: Summarize a research paper in 200 words.


# Agent: Summary Agent
## Final Answer:
This research paper on Convolutional Neural Networks (CNNs) delves into their architecture and diverse applications in visual perception. CNNs learn spatial hierarchies of features from images,


Validating summary length
Word count: 176
Summary exceeds 150 words
```

The Agent tries three attempts to fix it before it exits the program as shown below:

```
File /opt/anaconda3/lib/python3.12/site-packages/crewai/task.py:388, in Task._execute_core(self, agent, context, tools)
    386 if not guardrail_result.success:
    387     if self.retry_count >= self.max_retries:
--> 388         raise Exception(
    389             f"Task failed guardrail validation after {self.max_retries} retries. "
    390             f"Last error: {guardrail_result.error}"
    391         )
    393     self.retry_count += 1
    394     context = self.i18n.errors("validation_error").format(
    395         guardrail_result_error=guardrail_result.error,
    396         task_output=task_output.raw,
    397     )

Exception: Task failed guardrail validation after 3 retries. Last error: Summary exceeds 150 words. Word count: 176
```

If needed, one can also control the number of times the Agent must retry. This is defined using the `max_retries` parameter of the `Task` class:

This way, the Agent continues to reattempt until the error is fixed, i.e, we get return value in the format: `(True, task_output)` OR, until it has tried for a specified

number of retries in the `max_retries` parameter (which is 3, by default).

Let's look at another example now.

We know that structured outputs like JSON or CSV files must follow strict formatting. Although structured outputs work well and don't usually fall apart, AI-generated JSONs can sometimes:

- Contain syntax errors (missing commas, brackets, etc.).
- Be incomplete (missing required fields).
- Be invalid JSON altogether.

Let's build a guardrail for this.

Since we expect a strict JSON format, we define a Pydantic model for validation.

Now, we define a guardrail function that checks if the output is valid JSON and ensures all required fields are present. It returns errors when validation fails:

notebook.ipynb

```
import json
from typing import Tuple, Any

def validate_json_report(task_outp
    """""""
```

In the above code:

- If the JSON structure is correct, it returns `(True, task_output)`.
- it returns `(False, "Missing required fields...")`, prompting AI to retry.
- If the JSON syntax is incorrect, it returns `(False, "Invalid JSON format.")`.

Next, we define an Agent that specializes in generating structured research reports in JSON format.

notebook.ipynb

```
from crewai import Agent

# Create the AI Agent
research_report_agent = Agent(
    role="Research Analyst",
    goal="Generate structured JSON reports for research papers",
    backstory="You are an expert in structured reporting.",
    verbose=False)
```

Now, we create a task that instructs the agent to generate a valid JSON report:

notebook.ipynb

```
from crewai import Task

research_report_task = Task(
    description="Generate a structured JSON research report",
    expected_output="A JSON with 'title', 'summary', and 'key_findings'.",
    agent=research_report_agent,
    output_pydantic=ResearchReport,
    guardrail=validate_json_report,
    max_retries=3
)
```

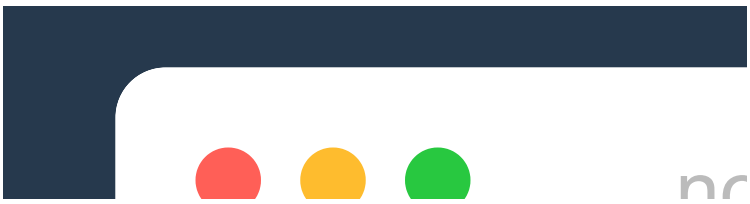
This task uses structured Pydantic output (`ResearchReport`) defined earlier and automatically validates AI-generated JSON using `validate_json_report` . We have also specify the maximum number of retries if the AI's output is incorrect.

Finally, we assemble our Crew, linking the Agent and Task.

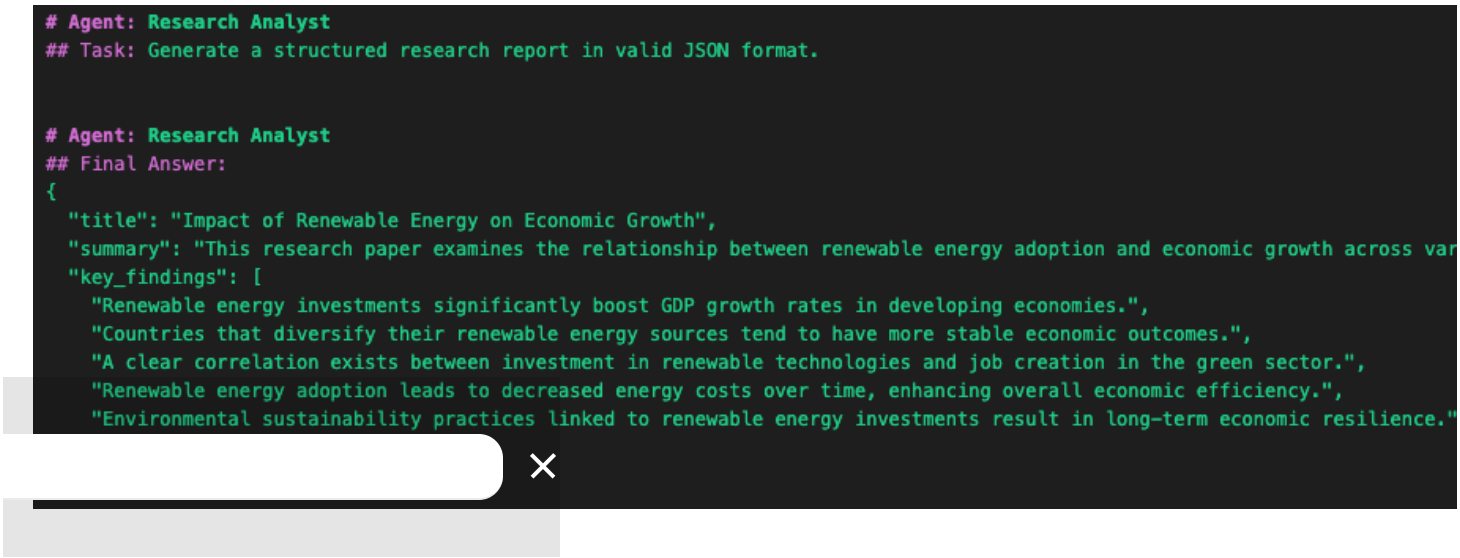
notebook.ipynb

```
from crewai import Crew

research_crew = Crew(
    agents=[research_report_agent],
    tasks=[research_report_task],
    verbose=True
)
```

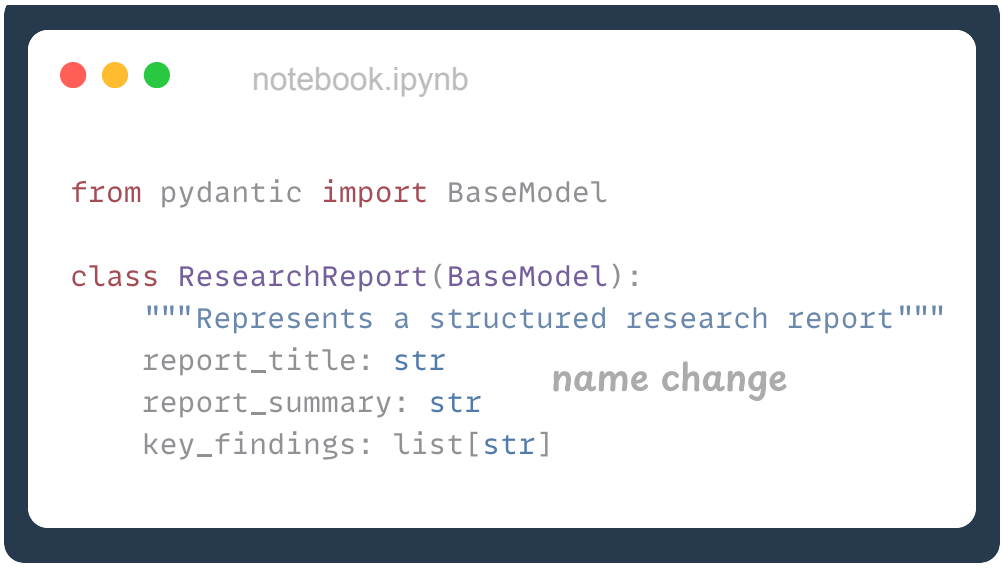
Finally, to execute the JSON report generation workflow, we run it:



This produces the following output, which is indeed correct.



At this step, as an exercise, alter the Pydantic model for validation we defined earlier to something like this:



Now re-run the entire workflow again (re-instantiate the agent, task, Crew, etc.) and run the Crew again. You will now notice the output of the guardrail method in the Crew's verbose output.

Let us know if you face any issues.

That said, let's learn about the best practices for using task guardrails below.

In your error messages, instead of just saying “Output is invalid”, provide clear instructions since these will be used by the Agent to improve the subsequent output. This helps the AI understand the issue and retry correctly.

Moreover, always apply guardrails where AI-generated errors can cause serious problems, such as:

- Data validation (JSON, CSV, financial reports) [if you have defined a Pydantic class, you will not need guardrails]
- Twitter posts, summaries). For instance, recall the Part 4 of this crash course where we built a Flow to create social content. On Twitter specifically, each tweet in a thread has a limit on the number of characters. In such cases, you can add a guardrail to ensure the tweet does not exceed the limit.
- Fact-checking outputs (news, research), etc.

Lastly, always ensure you have set a maximum retry limit. This prevents wasting API calls and compute power when AI struggles to meet requirements.

Referencing tasks and outputs

We have seen this before that when building multi-task AI workflows, it’s often necessary for one task to depend on another.

More specifically, tasks can reference previous task outputs to create context-aware AI workflows.

Below, let's explore:

- How CrewAI handles task outputs using the `TaskOutput` class.
- Ways to access different output formats (raw text, JSON, Pydantic models).
- How to make tasks depend on others.
- A complete example of a multi-step Crew that generates structured, interdependent AI outputs.

Firstly, understand that each task in CrewAI produces an output, which is automatically stored as a `TaskOutput` object.

Let's look at a simple example below:

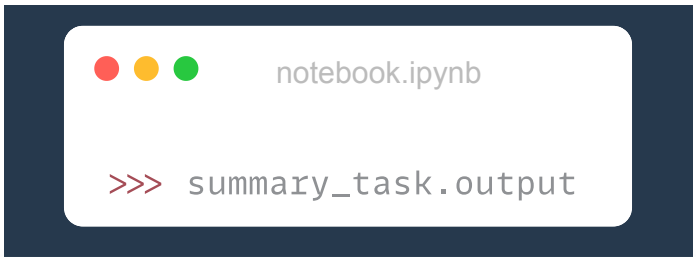
notebook.ipynb

```
from crewai import Task, Agent, Crew

summary_agent = Agent(
    role="Summary Agent",
    goal="Summarize the research paper 'CNNs' in 150 words.",
    backstory="You specialize in summarizing research topics.",
    verbose=True
)

summary_task = Task(
    description="Summarize a research topic in 150 words.",
    expected_output="A concise research summary 150 words.",
    agent=summary_agent,
)

summary_crew = Crew(
    tasks=[summary_task],
    agents=[summary_agent],
    verbose=True
)

result = summary_crew.kickoff()
```

This produces the following output:

```
# Agent: Summary Agent
## Task: Summarize a research paper in 150 words.


# Agent: Summary Agent
## Final Answer:
Convolutional Neural Networks (CNNs) are a class of deep learning models primarily used for image processing and co
```

Now, let's access the `output` attribute of the task object above—`summary_task`:




```
TaskOutput(description='Summarize a research topic in 150 words.',
           name=None,
           expected_output='A concise research summary 150 words.',
           summary='Summarize a research topic in 150 words....',
           raw="""Convolutional Neural Networks (CNNs) are a class of
                  deep learning models primarily used for image
                  processing and computer vision ...""",
           pydantic=None,
           json_dict=None,
           agent='Summary Agent',
           output_format=<OutputFormat.RAW: 'raw'>)
```

This includes several fields like:

- `description` → The task description
- `summary` → First 10 words of the description.
- `raw` → Raw AI-generated output.
- `pydantic` → A structured Pydantic model output (if any).
- `json_dict` → A JSON dictionary output.
- and more.

Now here's why this is useful to know:

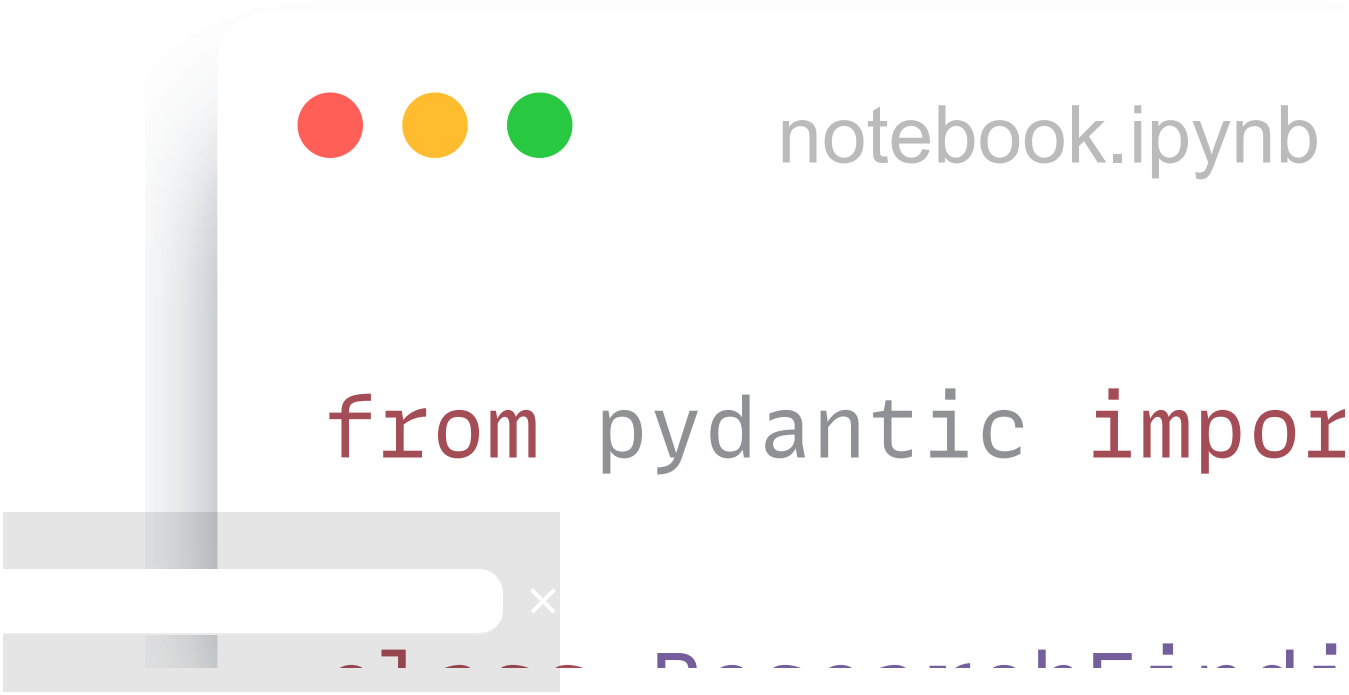
- You can reference earlier outputs instead of asking the AI to regenerate duplicate data.
- You can pass structured outputs (JSON/Pydantic models) to the next task for better control.

Let’s build a multi-step AI pipeline where:

- A research agent gathers AI industry trends.

- An analyst agent summarizes the findings.
- A blog writer agent generates a complete blog post using both research and summary as inputs.

Since structured data is more reliable, we define Pydantic models for task outputs.

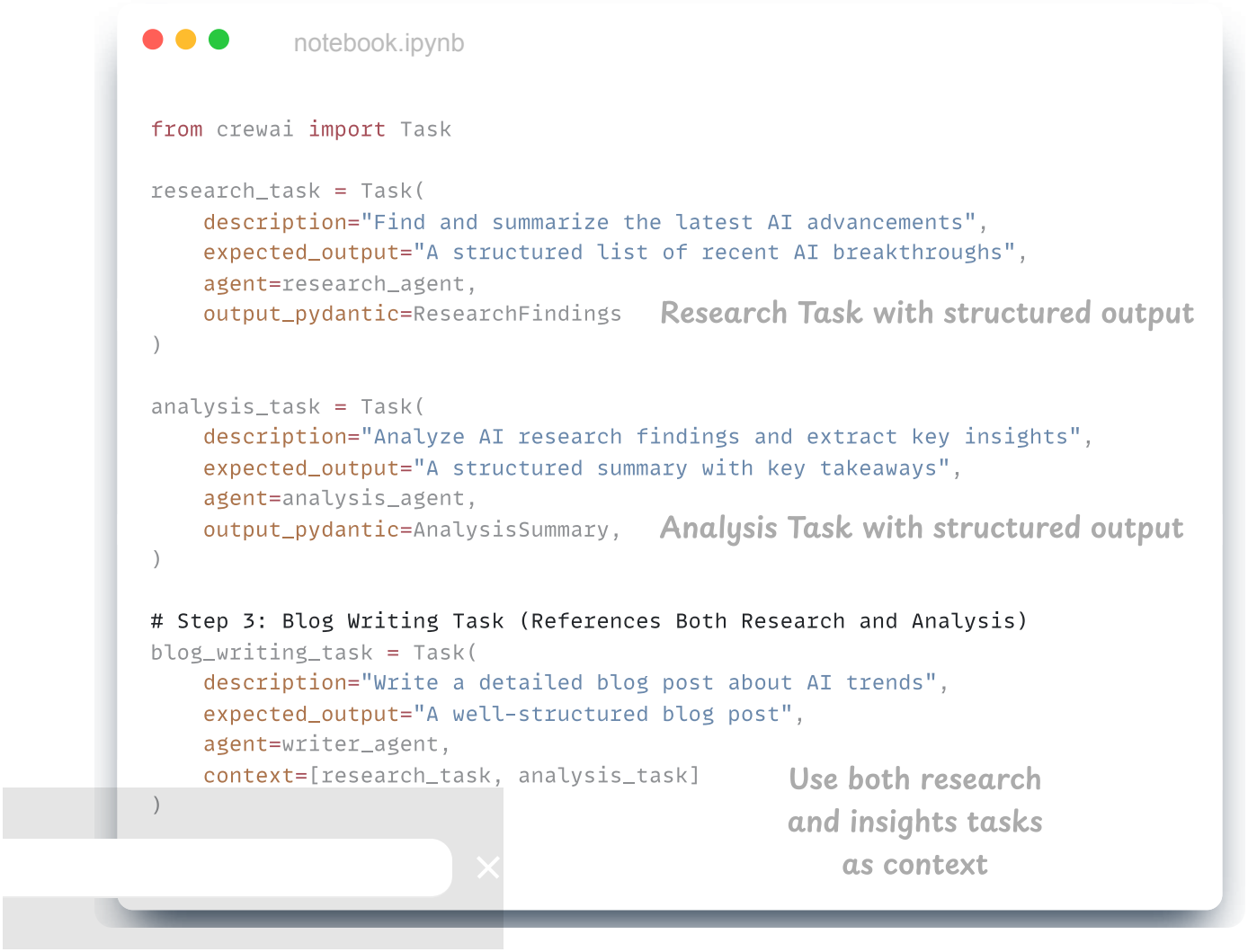


Next, we define our Agents and each AI agent is assigned a specific role:

notebook.ipynb

```python
from crewai import Agent

research_agent = Agent(
    role="AI Researcher",
    goal="Find and summarize the latest AI advancements",
    backstory="""You are an expert AI researcher who
                 stays up to date with the latest innovations.""",
    verbose=True,
)

analysis_agent = Agent(
    role="AI Analyst",
    goal="Analyze AI research findings and extract key insights",
    backstory="""You are a data analyst who extracts
                 valuable insights from research data.""",
    verbose=True,
)

writer_agent = Agent(
    role="Tech Writer",
                 well-structured blog post on AI trends",
                 ou are a technology writer skilled at
                 transforming complex AI research into
                 readable content.""",
    verbose=True,
)
```

Moving on, we create three interdependent tasks, where each task depends on outputs from previous tasks.



But if you look closely above, we have specified a `context` parameter in the Task definitions.

- The blog writing task depends on both research and analysis → `context=[research_task, analysis_task]`.

Moreover, while we could have explicitly specified that the analysis task depends on research findings using this → `context=[research_task]` (shown below), it is not needed since by default, a task always references the output of the previous task:

notebook.ipynb

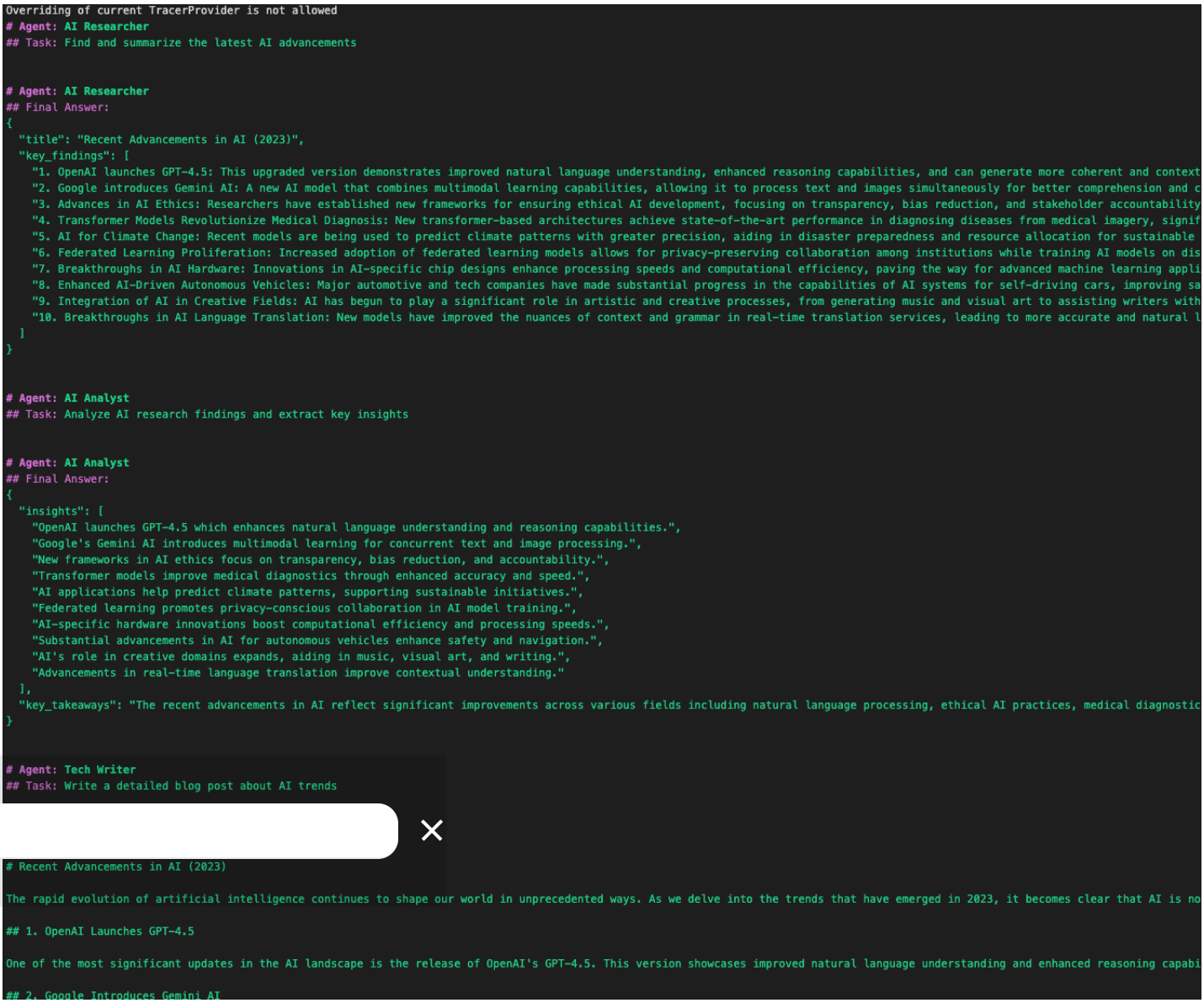
```
from crewai import Task

research_task = Task(
    description="Find and s
    expected_output="A stru
```

he AI crew and execute the workflow.

notebook.ipynb

```
from crewai import Crew

ai_research_crew = Crew(
    agents=[research_agent, analysis_agent, writer_agent],
    tasks=[research_task, analysis_task, blog_writing_task],
    verbose=True
)

result = ai_research_crew.kickoff()

print("\n --- Generated Blog Post ---")
print(result.raw)
```

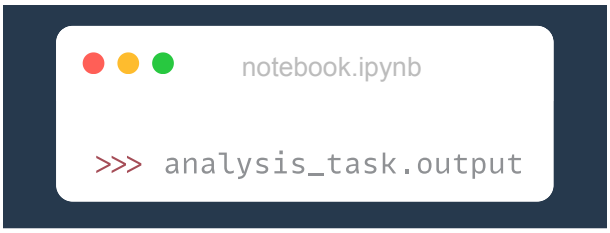
And we get the expected output:



In the above workflow:

- Research findings are gathered first.
- The analysis agent extracts insights.
- The blog writer creates a complete post using both previous outputs.

Also, as discussed earlier, if we want to explicitly access the output of any task, we can do so using the `output` attribute of the `task` object:




```
TaskOutput(description='Analyze AI research findings and extract key insights',
           name=None,
           expected_output='A structured summary with key takeaways',
           summary='Analyze AI research findings and extract key insights...',
           raw='{\n  "insights": ["OpenAI launches GPT-4.5 ...."}',
           json_dict=None,
           agent='AI Analyst',
           output_format=<OutputFormat.PYDANTIC: 'pydantic'>)
```

And to access the Pydantic output specifically, we can do this:

Essentially, with task referencing, we can:

- Chain multiple AI-generated outputs to build complex workflows.
- Avoid redundant AI generations by reusing previous task results.
- Pass structured JSON/Pydantic models between tasks for better accuracy.

This is also quite useful for debugging purposes to understand what tasks are producing what outputs.

Asynchronous Task Execution

In traditional task execution, each task runs sequentially—meaning one task must finish before the next begins. While this works well for dependent tasks, it can be inefficient when:

- Some tasks take longer to complete than others (e.g., web scraping, deep research).
- Tasks can run independently without affecting each other.
- We need to maximize efficiency by running tasks in parallel.

This is what asynchronous execution helps us with.

To exemplify, let's say we are building an AI-driven research workflow where:

- One agent researches AI breakthroughs.
- Another agent analyzes AI regulations.
- Both results are combined into a final report.

both the research agent and the analyst agent can work in parallel.

This is what async execution helps us with—Asynchronous execution allows multiple tasks to run in parallel, reducing overall execution time.

Let's look at a demo below.

To ensure clean and structured data, we define Pydantic models for each output.

notebook.ipynb

```python
from pydantic import BaseModel

class AIResearchFindings(BaseModel):
    """Represents structured research on AI breakthroughs."""
    title: str
    key_findings: list[str]

class AIRegulationFindings(BaseModel):
    """Represents structured research on AI regulations."""
    region: str
    key_policies: list[str]

class FinalAIReport(BaseModel):
    """Combines AI research & regulation analysis into a report."""
    executive_summary: str
    key_trends: list[str]
```

Next, we define three AI agents, each specializing in a different role.

notebook.ipynb

```python
from crewai import Agent

# Researcher for AI breakthroughs
research_agent = Agent(
    role="AI Researcher",
    goal="Find and summarize the latest AI breakthroughs",
    backstory="An expert AI researcher who tracks technological advancements.",
    verbose=True
)

# Analyst for AI regulations
regulation_agent = Agent(
    role="AI Policy Analyst",
    goal="Analyze global AI regulations and summarize policies",
    backstory="A government policy expert specializing in AI ethics and laws.",
    verbose=True
)

# Writer for the final AI report
writer_agent = Agent(
    role="AI Report Writer",
    goal="Write a structured report combining AI breakthroughs and regulations",
    backstory="A professional technical writer who crafts AI research reports.",
```

We now create two asynchronous tasks that run in parallel since they do not depend on each other. They gather research data on AI breakthroughs and regulations.

notebook.ipynb

```python
from crewai import Task

# Task 1: AI Breakthroughs Research (Asynchronous)
research_ai_task = Task(
    description="""Research the latest AI advancements
                   and summarize key breakthroughs.""",
    expected_output="A structured list of AI breakthroughs.",
    agent=research_agent,
    output_pydantic=AIResearchFindings,
    async_execution=True        Async execution
)

# Task 2: AI Regulation Analysis (Asynchronous)
research_regulation_task = Task(
    description="""Analyze the latest AI regulations
                   worldwide and summarize key policies.""",
    expected_output="A structured summary of AI regulations by region.",
    agent=regulation_agent,
    output_pydantic=AIRegulationFindings,
    async_execution=True        Async execution
)
```

Notice that in the above code, we have used `async_execution=True`. We did this because researching breakthroughs and regulations are independent, they can run at the same time.

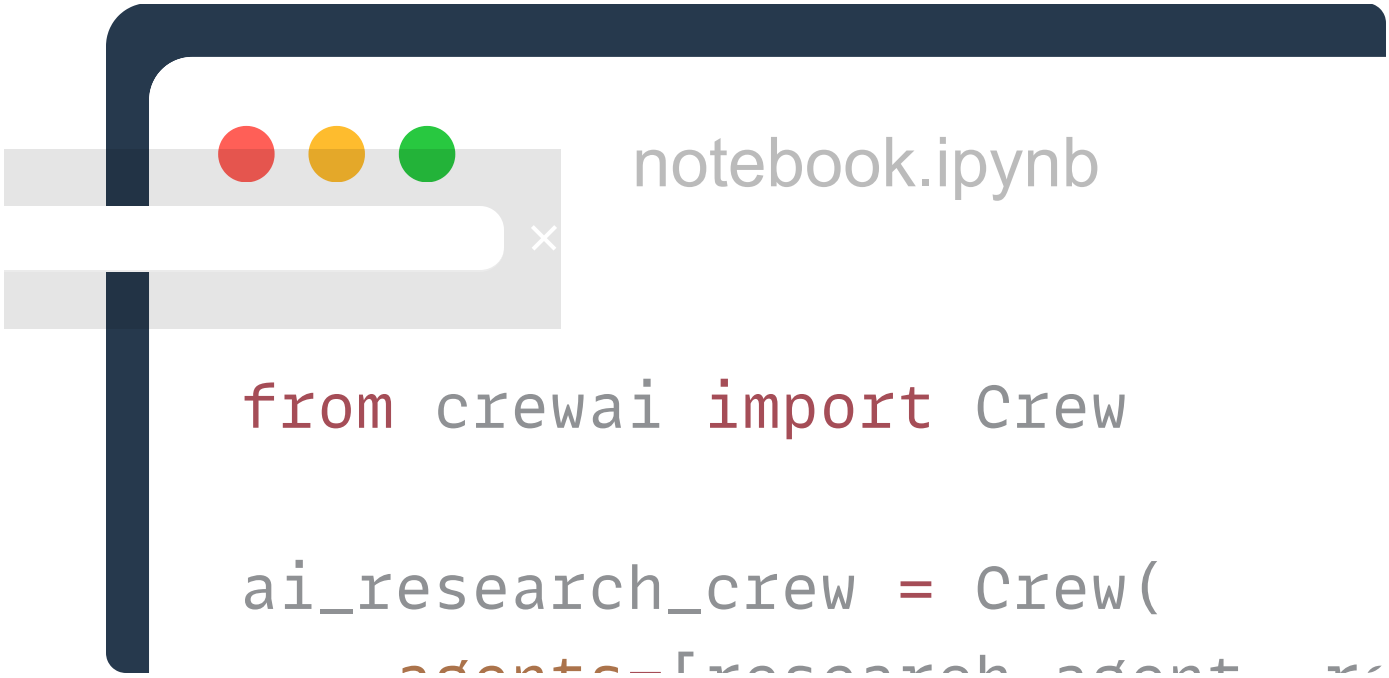
Once both research tasks are complete, the AI Report Writer must reference outputs from both research tasks and create the data into a structured report. This is implemented below:

notebook.ipynb

```
# Task 3: Generate AI Research Report
generate_report_task = Task(
    description="Write a report summarizing AI breakthroughs and regulations.",
    expected_output="A final AI report summarizing both aspects.",
    agent=writer_agent,
    output_pydantic=FinalAIReport,
    context=[research_ai_task, research_regulation_task]
)
```

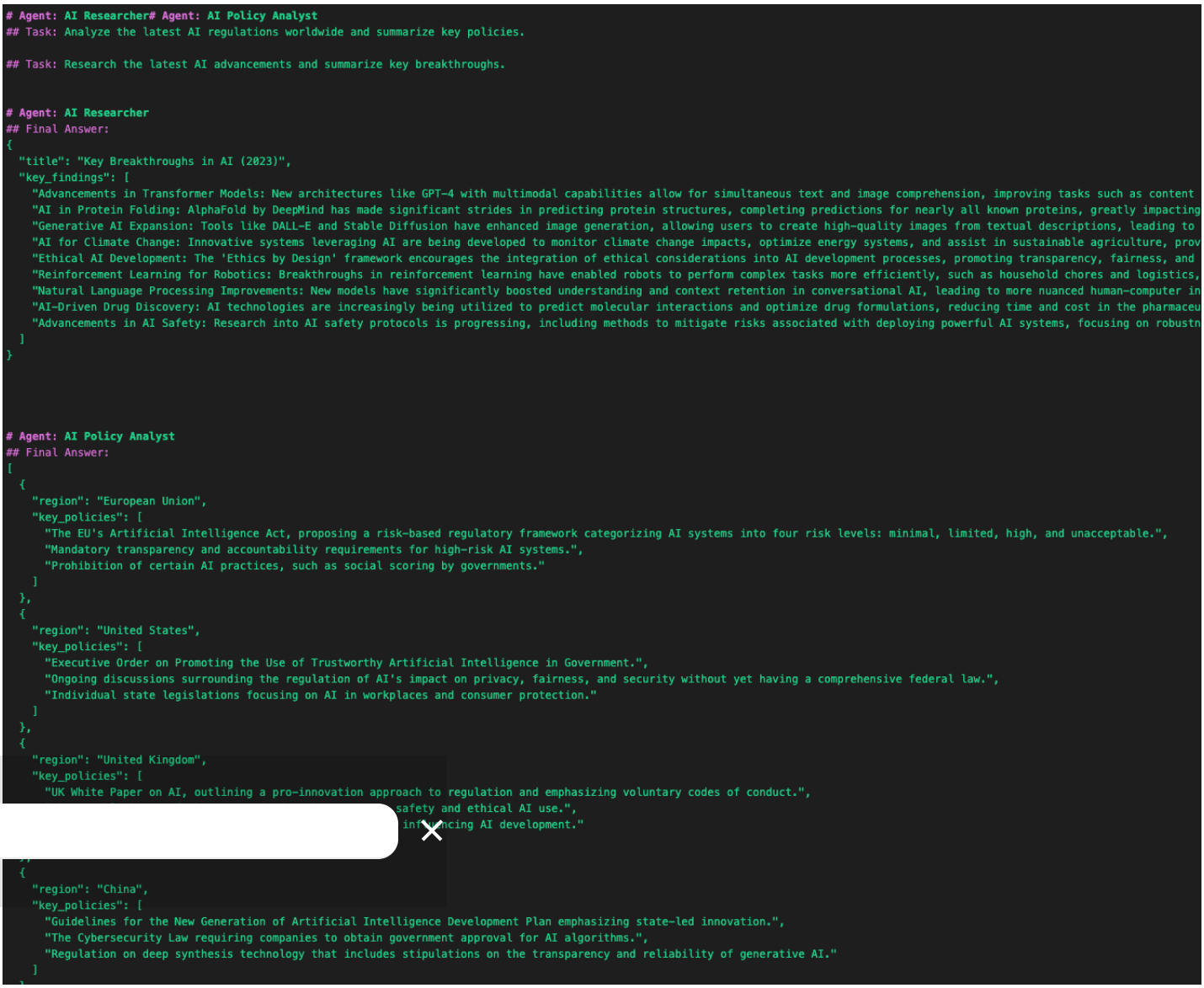
Uses the output of first two tasks

Now, we assemble the AI crew and execute the workflow.

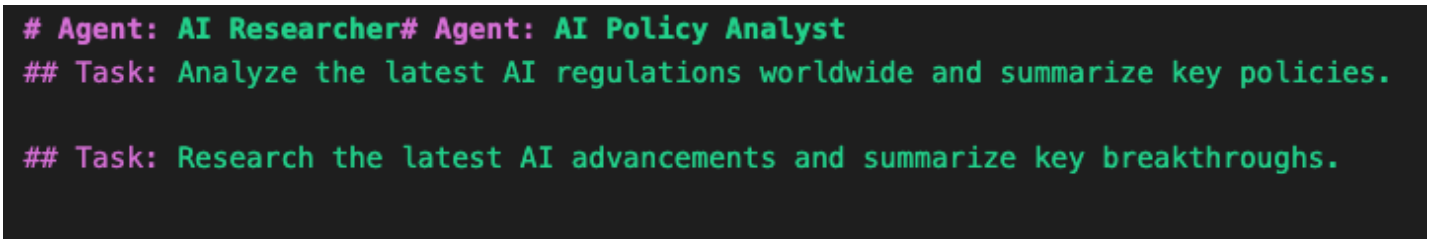


Done!

This produces the following output.



Notice that the tasks for both the AI Researcher Agent and the AI Policy Analyst Agent have been executed simultaneously.



Had we not used `async_execution=True` earlier, we would have first observed the output of the AI Researcher Agent, and then the task of the AI Policy Analyst must have been initiated.

But since the task of the AI Policy Analyst was initiated before the output of the AI Researcher Agent was produced, this shows that both ran independently of each other.

We can also verify this by setting the in the following to tasks and rerunning the workflow.

notebook.ipynb

```
from crewai import Task

# Task 1: AI Breakthroughs Research (Asynchronous)
research_ai_task = Task(
    description="""Research the latest AI advancements
                   and summarize key breakthroughs.""",
    expected_output="A structured list of AI breakthroughs.",
    agent=research_agent,
    output_pydantic=AIResearchFindings,
    async_execution=False       Async execution disabled
)

# Task 2: AI Regulation Analysis (Asynchronous)
research_regulation_task = Task(
    description="""Analyze the latest AI regulations
                   worldwide and summarize key policies.""",
    expected_output="A structured summary of AI regulations by region.",
    agent=regulation_agent,
    output_pydantic=AIRegulationFindings,
    async_execution=False       Async execution disabled
)
```

This produces the following verbose execution log, and it is clear that the AI Policy Analyst's task was initiated after the output of the AI Researcher Agent was produced.

We hope this clarifies the difference between synchronous and asynchronous task executions in agentic systems.

Adding callbacks

In many AI-driven workflows, you might need to trigger additional actions once a task is completed.

For example, once a task finishes, you may want to:

- Store results in a database or file for further processing
- Trigger another process (like running a Python script)
- or anything else.

CrewAI provides task callbacks to handle such scenarios—allowing you to execute a function immediately after a task finishes.

Let’s consider building a news monitoring system where:

- An AI researcher agent finds the latest AI news.
- A callback function sends an alert when research is completed.

We first create an AI agent responsible for researching the latest AI news.

Next, we define a callback function that automatically executes when the research task completes.

notebook.ipynb

Callback function to notify the team after research completion

```python
def notify_team(output):

    print(f"""Task Completed!
              Task: {output.description}
              Output Summary: {output.summary}""")

    with open("latest_ai_news.txt", "w") as f:
        f.write(f"Task: {output.description}\n")
        f.write(f"Output Summary: {output.summary}\n")
        f.write(f"Full Output: {output.raw}\n")

    print("News summary saved to latest_ai_news.txt")
```

Now, we create a task for the Agent created above and specify a callback function using the `callback` parameter.

notebook.ipynb

```python
from crewai import Task

research_news_task = Task(
    description="Find and summarize the latest AI breakthroughs",
    expected_output="A structured summary of AI news headlines.",
    agent=research_agent,
    callback=notify_team
)
```

Callback function

This way, the research task will execute normally. But once the task is complete, the callback function will automatically run. The news summary will be saved in a text file, and a notification will be printed.

Now, we assemble the CrewAI workflow and execute it.

notebook.ipynb

```python
from crewai import Crew

ai_news_crew = Crew(
    agents=[research_agent],
    tasks=[research_news_task],
    verbose=False
)

result = ai_news_crew.kickoff()
```

Here's the terminal output after crew's execution.

```
Task Completed!
            Task: Find and summarize the latest AI breakthroughs from the last week.
            Output Summary: Find and summarize the latest AI breakthroughs from the last...
News summary saved to latest_ai_news.txt
```

And we get this output in the `latest_ai_news.txt` file:

latest_ai_news.txt

```
Task: Find and summarize the latest AI breakthroughs from the last week.
Output Summary: Find and summarize the latest AI breakthroughs from the last...
Full Output: 1. **Google AI Unveils Gato: A Multi-Modal Agent Capable of 600 Tasks**
   Google has introduced Gato, a revolutionary AI model that demonstrates multi-tasking capabilities across various domai

2. **OpenAI Enhances ChatGPT with New Features and Plugins**
   OpenAI has rolled out several updates to ChatGPT, adding new plugins that enable users to access real-time data, perfo

3. **NVIDIA Launches Next-Gen GPUs to Power AI Research**
   NVIDIA has launched its latest line of GPUs, specifically optimized for training large AI models. The new hardware boa

4. **Meta's AI Research Achieves Breakthrough in Natural Language Understanding**
   Meta's latest research has introduced a model that improves natural language understanding, which could enhance the ac

5. **AI-Powered Drug Discovery Program Shows Promising Results**
   A collaboration between biopharmaceutical companies and AI firms has yielded promising results in identifying new drug

6. **Ethics in AI: New Guidelines Proposed by International Coalition**
   An international coalition of AI researchers and ethicists has proposed new guidelines aimed at ensuring ethical pract

These breakthroughs not only showcase the rapid advancements in AI technology but also highlight the importance of ethica
```

While this is a simple use case, you can extend callbacks to much more advanced use cases.

Conclusion, takeaways, and next steps

With that, we come to the end of Part 5 of the Agents crash course.

In this part, we explored advanced techniques that make AI workflows in CrewAI more structured, efficient, and reliable.

Instead of simply executing tasks in a linear sequence, we introduced key mechanisms that enhance control, flexibility, and performance.

Here are some key takeaways:

- Enforcing guardrails for reliable AI outputs:
    - We used guardrails to validate AI-generated outputs, ensuring they meet specific constraints.
    - This prevents issues like overly long responses, incorrect formatting, or missing key details.
    - We also saw how AI can self-correct when guardrails detect errors, improving output quality.
- Referencing other tasks:
    - Instead of working in isolation, agents can dynamically reference outputs from previous tasks.
    - This allows for more context-aware decision-making, where tasks build on prior results.
    - We demonstrated how multiple research tasks could contribute to a final AI-generated report.
- Optimizing workflows with asynchronous execution
    - Tasks don’t always need to run sequentially—some can be executed in parallel to save time.

    - We enabled asynchronous execution, allowing AI agents to work simultaneously where possible.
    - This improves efficiency, especially in workflows involving multiple independent tasks.
- Using Callbacks to do actions on task completion
    - With callbacks, we can trigger actions after a task completes.
    - This is useful for logging, post-processing, sending notifications, or triggering follow-up tasks.
    - We implemented a system where a callback function logs AI research results once generated.

> 💡 As a key takeaway, always think of your agentic system as a structured design—just as you would in a real-world human workflow. When designing an AI-driven system, ask yourself:
>
> - How would I structure this process if I were working with a team of
> - Which tasks need to be done first, and which can be done in parallel?
> - Who (or which AI agent) should oversee and validate the work?
> - Which Agents should be enforced with guardrail?
> - and more.
>
> Once you have this layout on paper, it becomes immensely easier to extend this to an implementation.

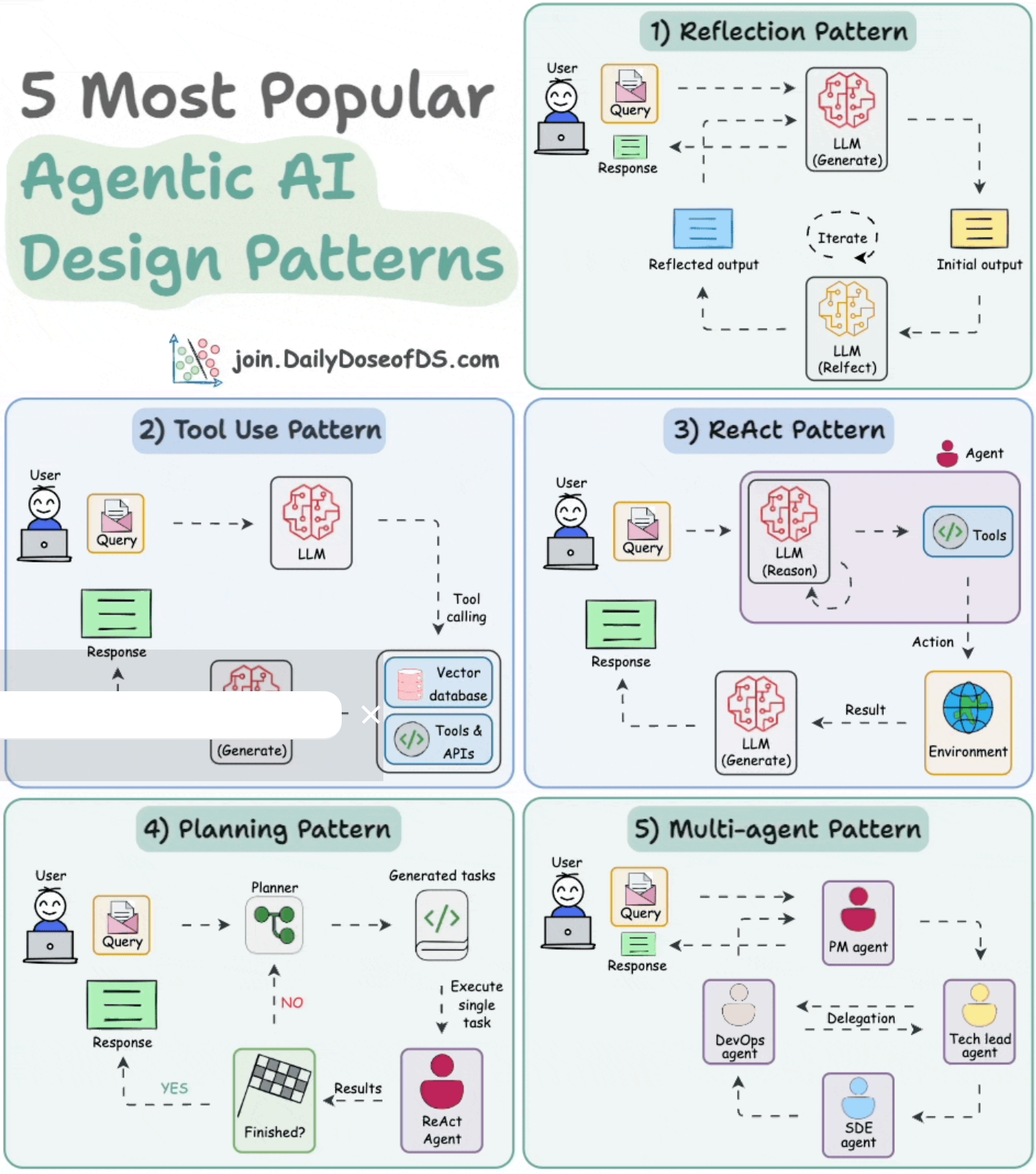
What's Next in Part 6?

In Part 6, we will continue this by exploring:

- Human-in-the-loop interactions → Allowing users to review and refine AI-generated outputs.
- Hierarchical processes → Structuring AI workflows where manager agents oversee task execution.
- Multimodal agents → Enabling agents to process both text and images within the same workflow.

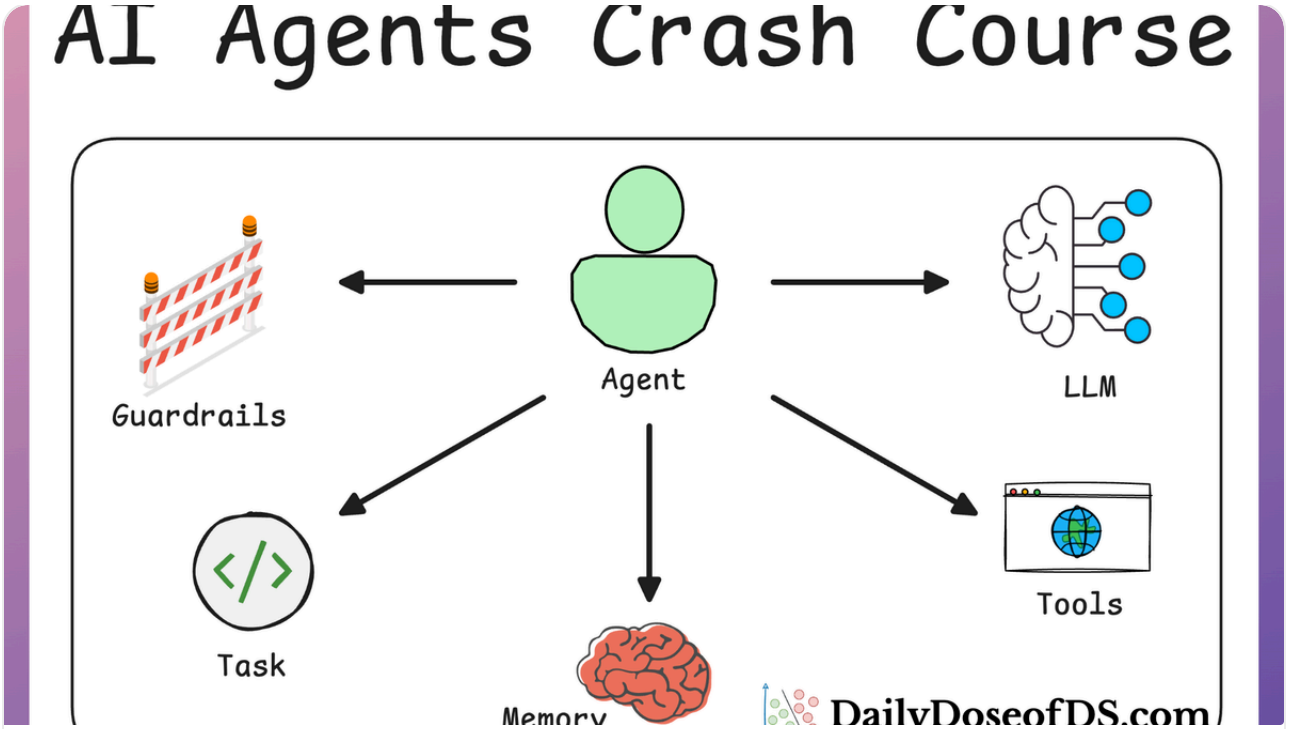
Also, we aren't done with the Agent crash course yet. In the upcoming parts, we have several other advanced agentic things planned for you:

- Building production-ready agentic pipelines that scale.
- Agentic RAG (Retrieval-Augmented Generation) – combining RAG with AI agents.
- Optimizing agents for real-world applications in business and automation.
- Building Agents around the Agentic patterns depicted below:



- and many many more.

Read the next part of this crash course here:



Advanced Techniques to Build Robust Agentic Systems (Part B)

AI Agents Crash Course—Part 6 (with implementation).

Daily Dose of Data Science • Avi Chawla

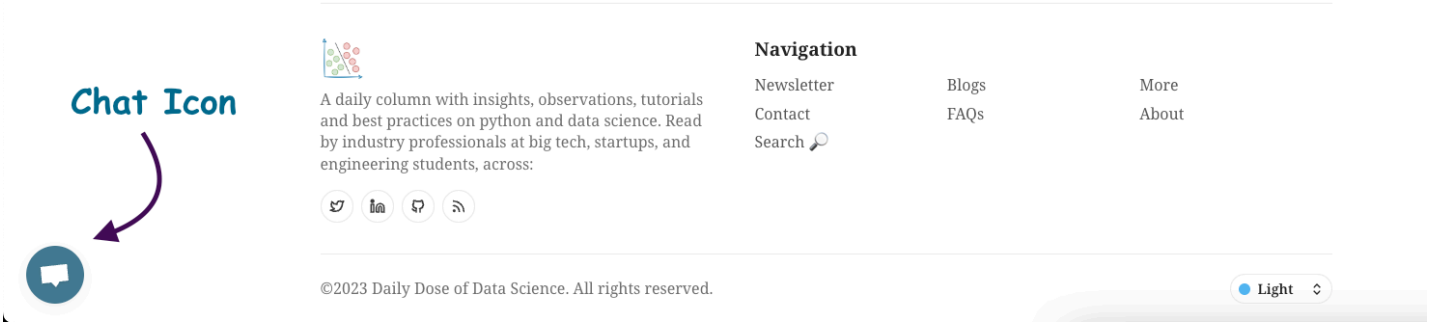
As always, thanks for reading!

Any questions?

Feel free to post them in the comments.

Or

If you wish to connect privately, feel free to initiate a chat here:



Connect via chat

Agents LLMs AI Agent Crash Course

Share this article

Read next

MCP Jun 15, 2025 • 22 min read

The Full MCP Blueprint: Building a Full-Fledged MCP Workflow using Tools, Resources, and Prompts

Model context protocol crash course—Part 4.

Avi Chawla, Akshay Pachaar

Agents Jun 8, 2025 • 20 min read

The Full MCP Blueprint: Building a Custom MCP Client from Scratch

Model context protocol crash course—Part 3.

Avi Chawla, Akshay Pachaar

Agents Jun 1, 2025 • 22 min read

The Full MCP Blueprint: Background, Foundations, [illegible] ractical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar

A daily column with insights, observations, tutorials and best practices on python and data science. Read by industry professionals at big tech, startups, and engineering students, across:

Navigation

Sponsor Newsletter More Contact