Mar 2, 2025

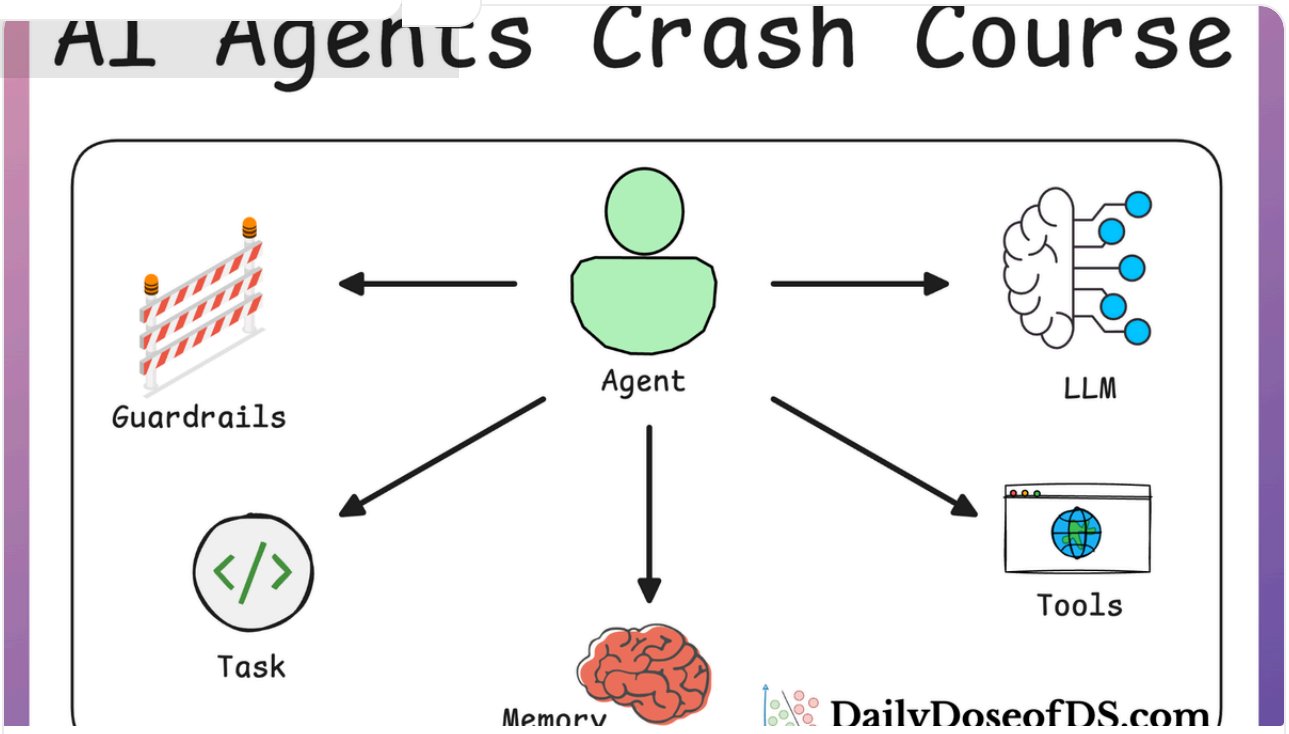
Building Flows in Agentic Systems (Part B)

AI Agents Crash Course—Part 4 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

In the previous part, we explored how to build structured AI workflows using CrewAI Flows.



AI Agents Crash Course—Part 3 (With Implementation)

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

We covered 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.

This allows for the design and implementation of multi-step processes that leverage the full potential of CrewAI's capabilities.

However, in real-world applications, a single Crew is often not enough.

As we shall see shortly, complex workflows require multiple specialized Crews working together to complete different aspects of a larger process.

focus on multi-crew flows, where several Crews operate in parallel or sequentially to achieve a common goal. We will walk through real-world use cases, breaking down how different Crews collaborate, share information, and optimize workflow execution.

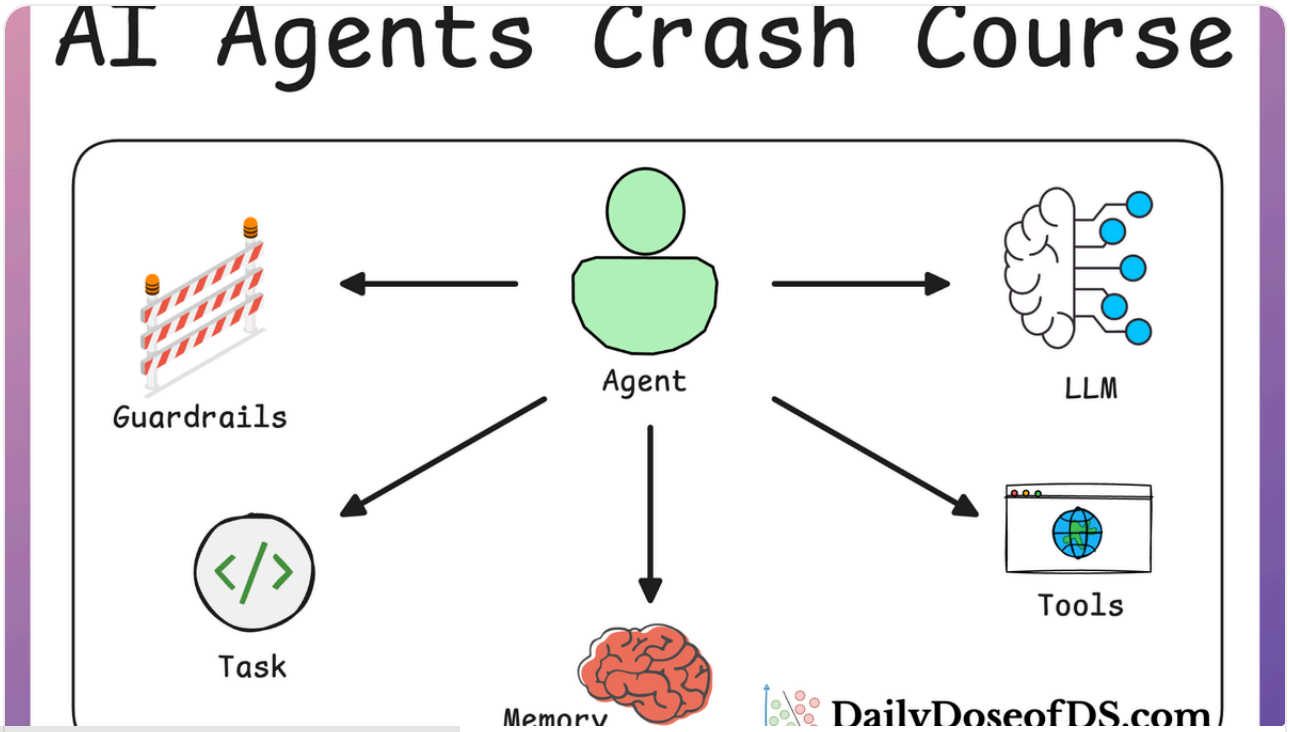
By the end of this part, you will understand how to:

- Design and implement flows that combine multiple Crews efficiently.
- Use Crew dependencies to control task execution and coordination.

And of course, everything will be supported with proper implementations like we always do!

Let’s begin by recapping the fundamental principles of multi-crew architectures and their advantages.

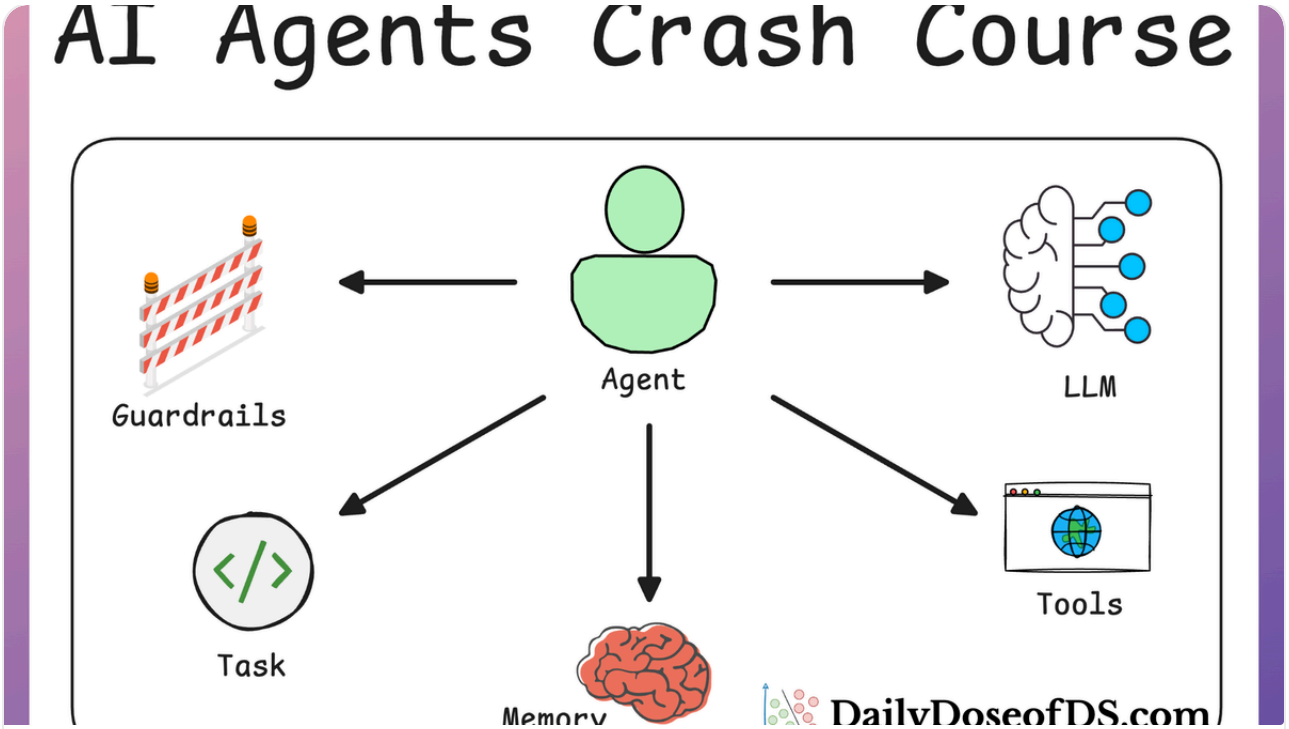
If you haven't read Part 1, Part 2 and Part 3 yet, it is highly recommended to do so before moving ahead.



AI Agents Crash Course—Part 1 (With Implementation)

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

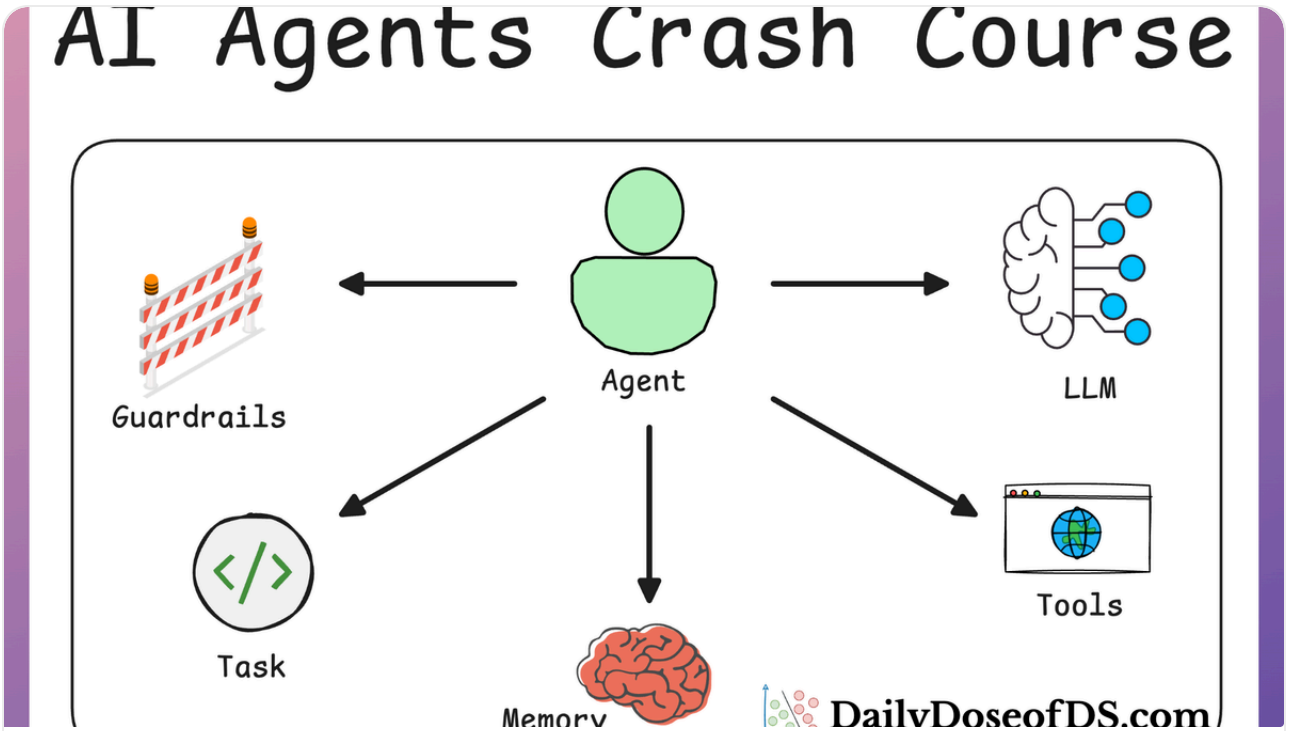
Daily Dose of Data Science • Avi Chawla



AI Agents Crash Course—Part 2 (With Implementation)

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

ience • Avi Chawla



AI Agents Crash Course—Part 3 (With Implementation)

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

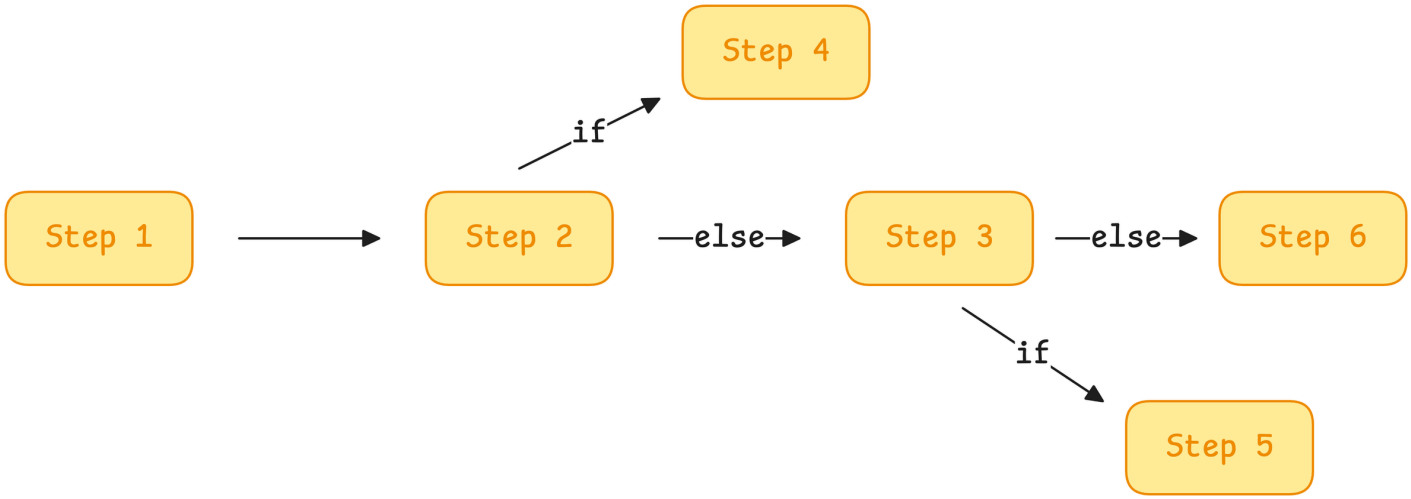
Recap

💡 Feel free to skip to the next section if you remember Flows and the technical details we discussed last time.

Why Flows?

As we learned in the previous parts, in traditional software development, workflows are meticulously crafted with explicit, deterministic logic to ensure predictable outcomes.

- IF A happens → Do X
- IF B happens → Do Y
- Else → Do Z



Of course, there's nothing wrong, and this approach excels in scenarios where tasks are well-defined and require precise control.

However, as we integrate Large Language Models (LLMs) into our systems, we encounter tasks that benefit from the LLMs' ability to reason, interpret context, and handle ambiguity—capabilities that deterministic logic alone cannot provide.

For instance, consider a customer support system.

Traditional logic can efficiently route queries based on keywords, but understanding nuanced customer sentiments or providing personalized responses requires the interpretative capabilities of LLMs.

Nonetheless, allowing LLMs to operate without constraints can lead to unpredictable behavior at times.

Therefore, it's crucial to balance structured workflows with AI-driven autonomy.

Flows help us do that.

Essentially, Flows enable developers to design workflows that seamlessly integrate deterministic processes with AI's adaptive reasoning.

By structuring interactions between traditional code and LLMs, Flows ensure that while AI agents have the autonomy to interpret and respond to complex inputs, they do so within a controlled and predictable framework.

In essence, Flows provide the infrastructure to harness the strengths of both traditional software logic and AI autonomy, creating cohesive systems that are both reliable and intelligent.

It will become easier to understand them once we get into the implementation so let's jump to that now!

Installation

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.

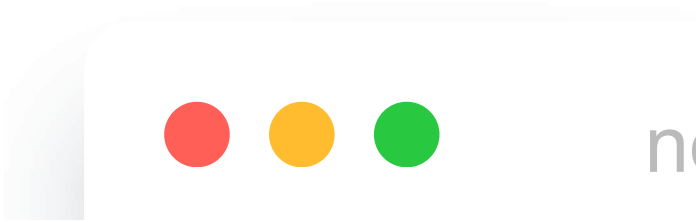
Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling...

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:

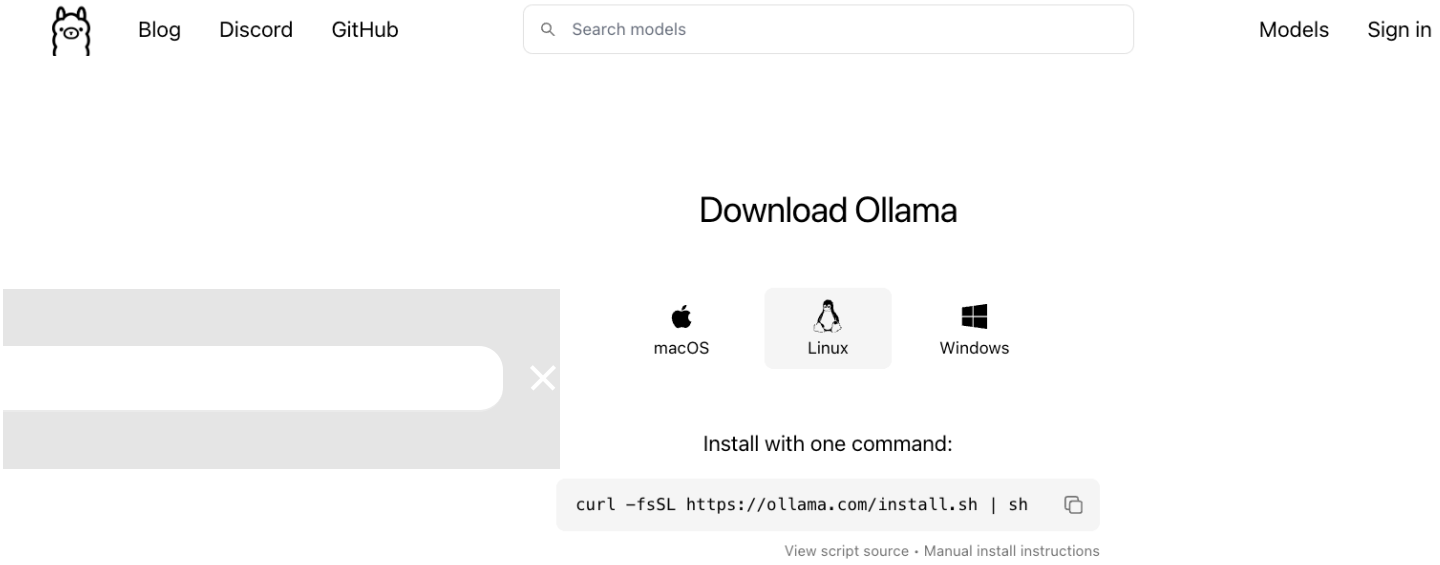
- OpenAI
- Gemini
- Groq
- Azure
- Fireworks AI
- SambaNova
- and many more.

> 💡 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 some 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.

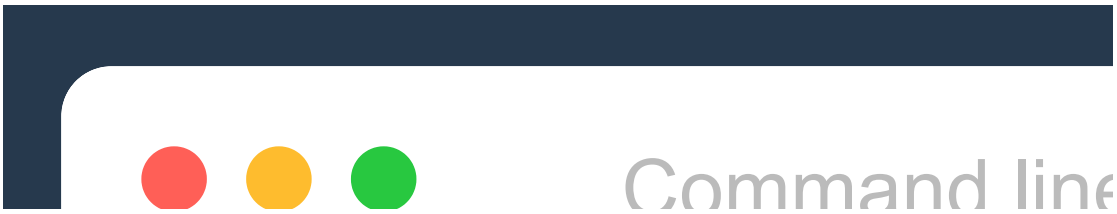
To set up OpenAI, create a `.env` file in the current directory and specify your OpenAI API key as follows:

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

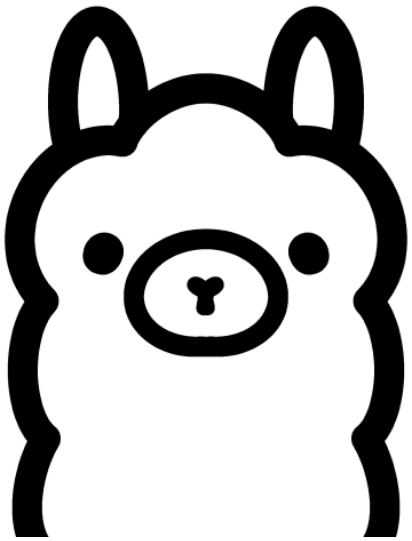
- Go to **Ollama.com**, select your operating system, and follow the instructions.



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

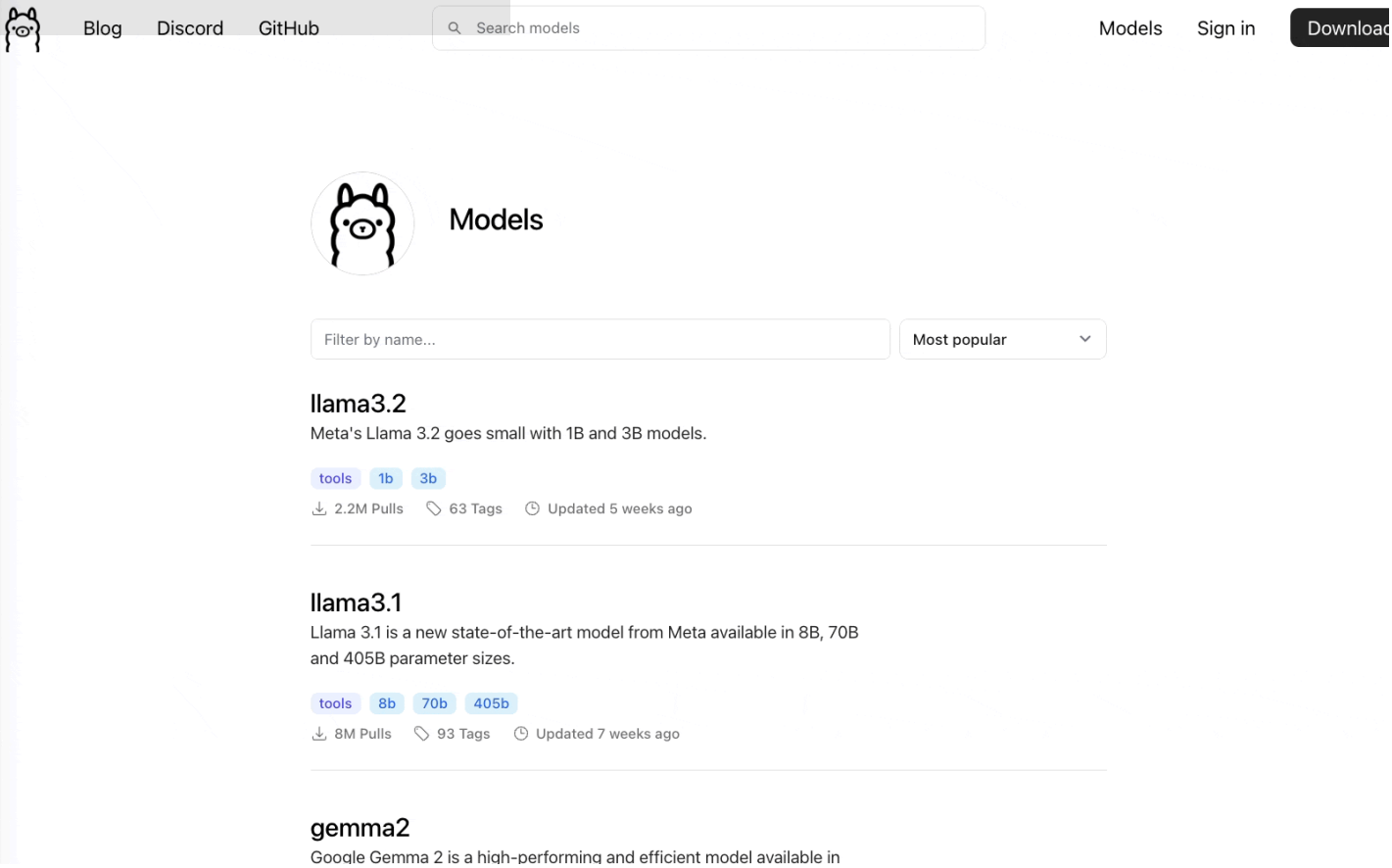


- Ollama supports a bunch of models that are also listed in the model library:

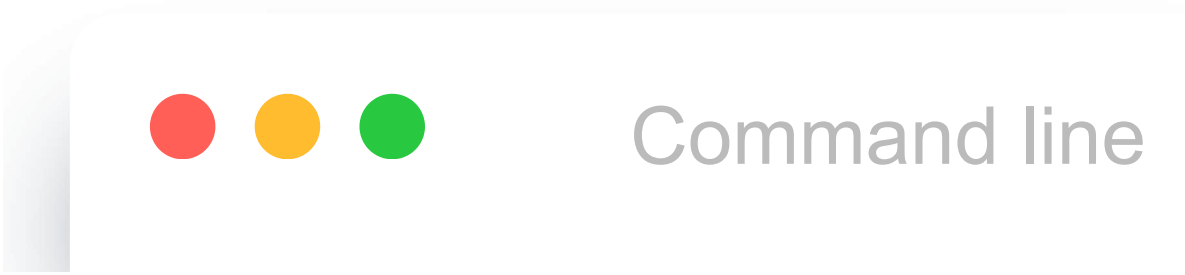


library

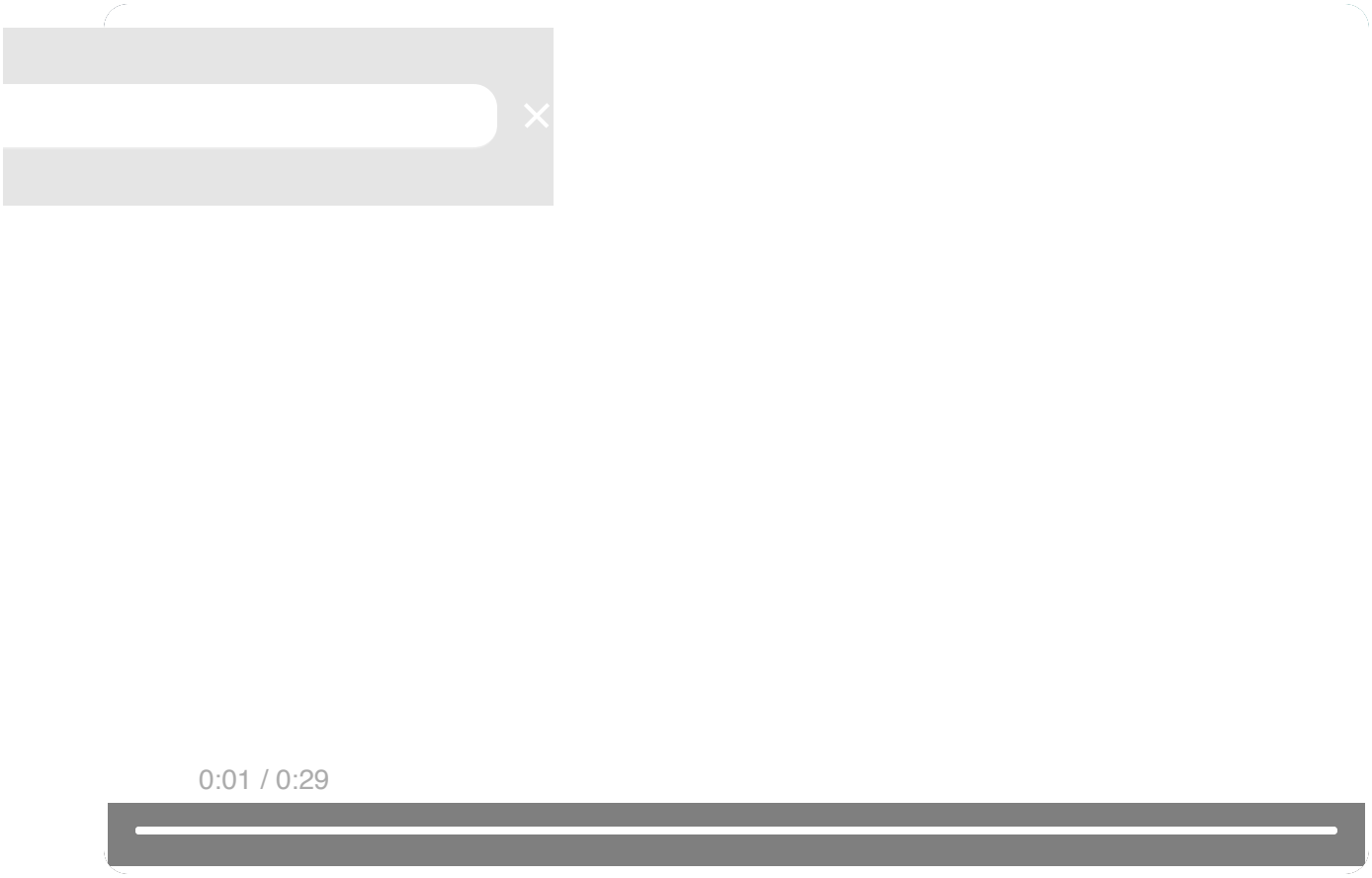
Get up and running with large language models.



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 our Flows.

Technical recap

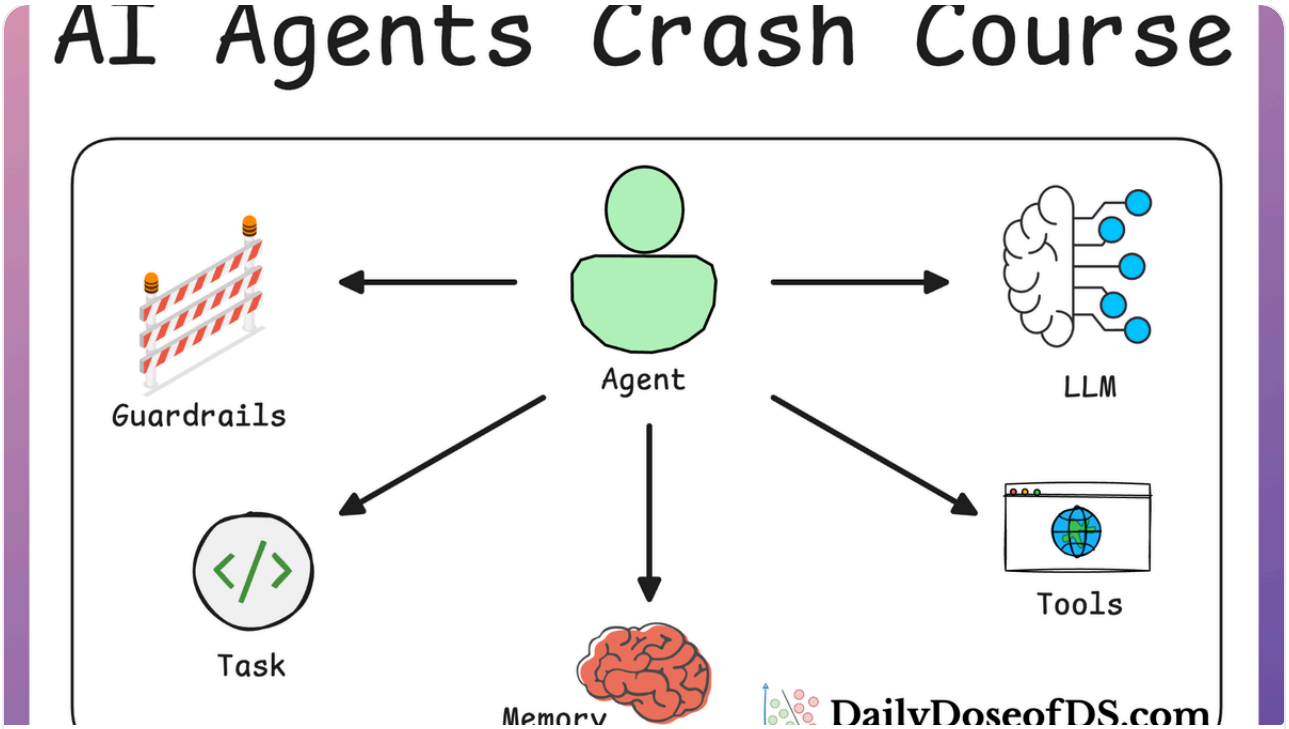
technical details we discussed last time.

Also, you can download the code for this article below:

AI Agent Crash course Part 4

AI Agent Crash course Part 4.zip • 125 KB

If you want to learn them in detail and haven't read Part 3 yet, we recommend doing that before diving ahead:

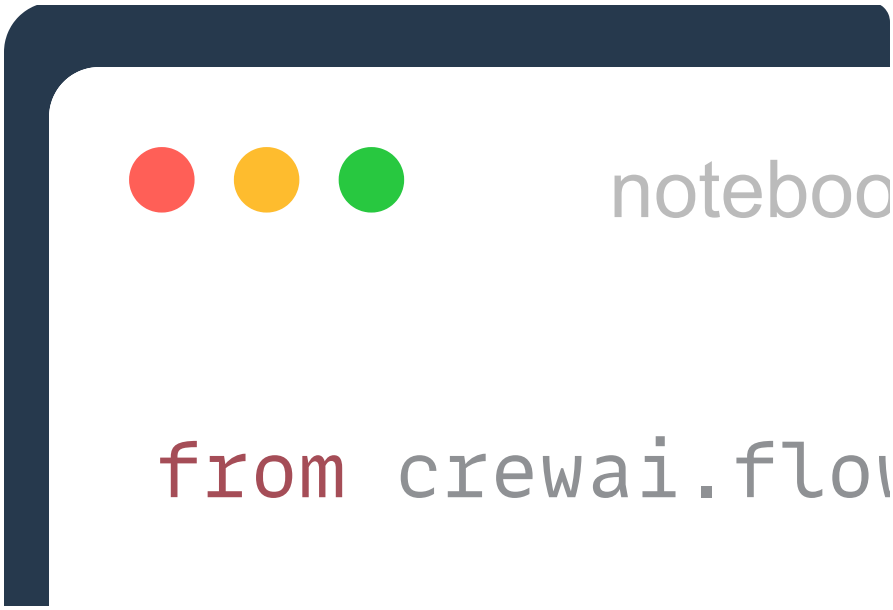


AI Agents Crash Course—Part 3 (With Implementation)

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

Daily Dose of Data Science • Avi Chawla

1) The `@start` decorator marks the first method that begins execution in a Flow. This function runs automatically when the Flow starts.



2) A method with `@listen` runs after another method completes.

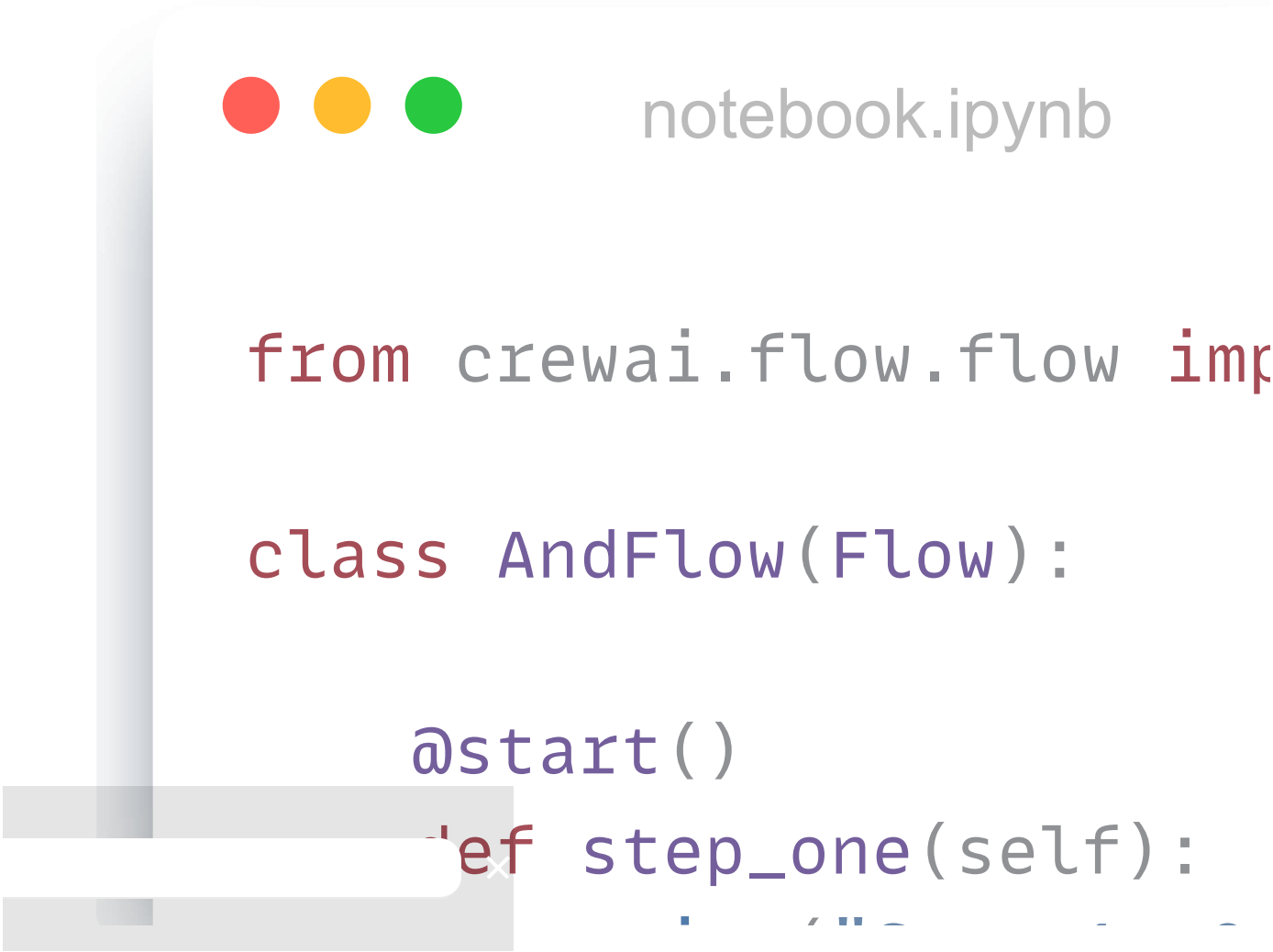
In the above code:

- `first_task` runs first.
- Once it finishes, `second_task` listens to its output and executes next.

3) The `or_()` function allows a task to execute when at least one of its dependencies is completed.

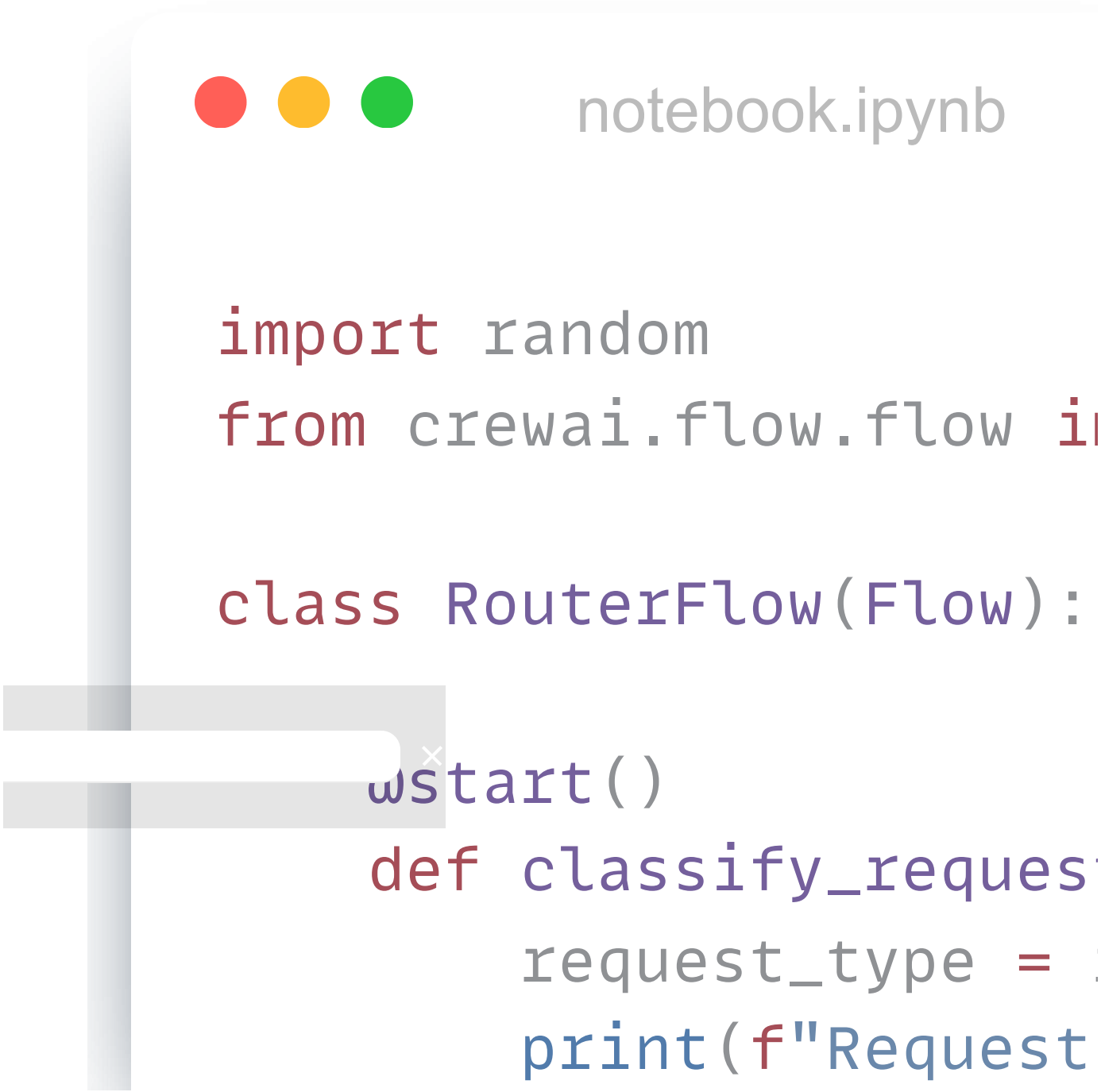
In the above code, if either `fetch_from_api` or `read_from_db` completes, `process_data` is triggered.

4) The `and_()` function ensures that a task runs only after all dependencies have been completed.



In the above code, `final_step` runs only after both `step_one` and `step_two` finish execution.

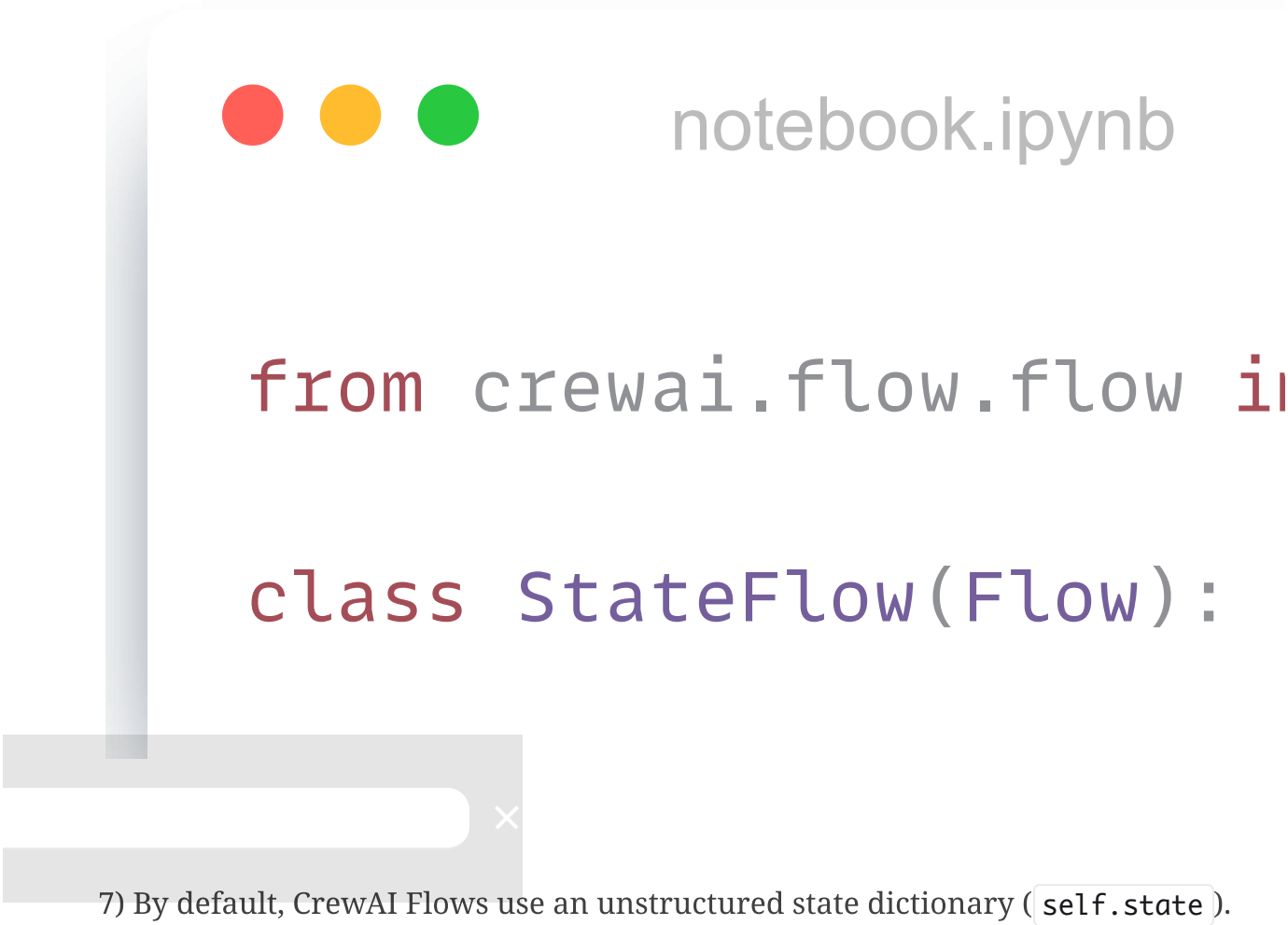
5) The `@router()` decorator directs execution based on conditional logic.



In the above code:

- `classify_request` randomly sets a request type (`urgent` or `normal`).
- The router function directs execution to either `urgent_handler` or `normal_handler`.

6) Each Flow has a state dictionary (`self.state`), which persists values across steps. The state persists values between tasks, allowing dynamic updates.



7) By default, CrewAI Flows use an unstructured state dictionary (`self.state`). However, for better data validation, type safety, and maintainability, we can use structured state management with `pydantic` . Structured states define a fixed schema for storing and managing state values.

In the above code:

- The state is strictly defined using `pydantic`.
- Type safety ensures that `count` is always an integer.
- If an undefined attribute is accessed, CrewAI raises an error instead of silently allowing unexpected behavior.

Now that we have covered task execution, flow control, and state management, we can move on to building complex multi-crew Flows.

In the next section, we will apply these principles to real-world use cases and learn how to coordinate multiple Crews within a Flow.

Demo 1: Social Media Content Writer Flow

The following video walks you through building a multi-agent app that can automatically write and publish social media content.

Here's our tech stack:

- CrewAI to build an Agentic workflow.
- **FireCrawl** for web scraping.

The entire multi-agent system is totally hands-off and automated, and we heavily used CrewAI flows in this demo.

Here's how it works:

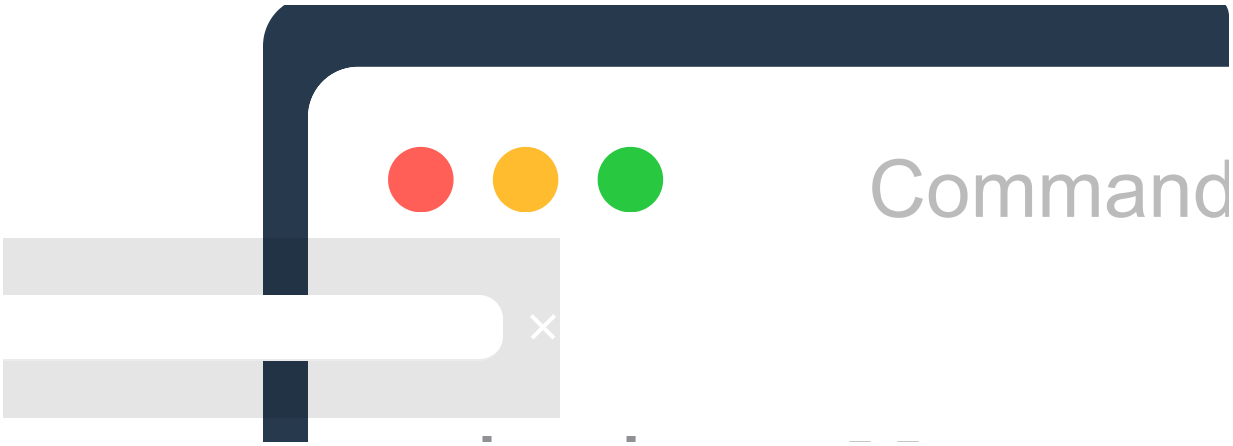
- We provide a link to a blog (in our case, it's our newsletter issue).
- We use **Firecrawl** to scrape the newsletter (with images) and save it as a markdown.
- Since there are multiple agents, one agent has access to our existing social content to understand our writing style.

- Next, since we publish content on LinkedIn and X, we built two routers in this agentic workflow. Based on a trigger, another agent gets executed to write a ready-to-publish draft.
- Finally, we save the content plan as a structured JSON file.

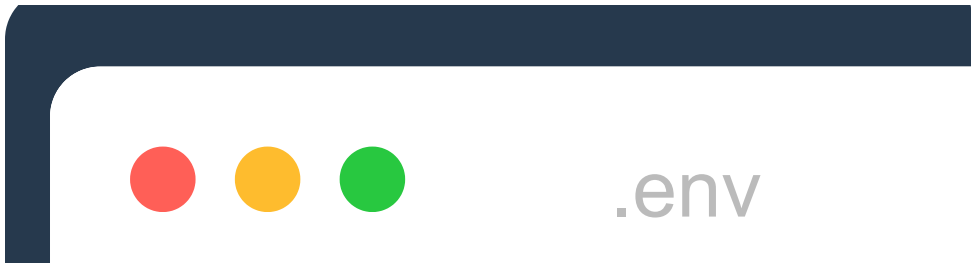
Let's look at the full end-to-end implementation below.

Setup

Before running the flow, install the necessary dependencies:



Also, create a `.env` file and add your API keys:



Define the LLM

We are using GPT-4o for all AI-generated outputs:

Alternatively, you can replace it with a **local Ollama model** (make sure that your Ollama server is running and you have the model available locally as follows):

Sample files

To ensure that the AI-generated content follows a consistent writing style, the assets folder contains two sample files:

- `example_threads.txt` → Contains examples of well-structured Twitter (X) threads.
- `example_linkedin.txt` → Contains samples of LinkedIn posts with proper formatting, tone, and structure.

This is already available in the code folder shared above.

Configure AI Agents and Tasks

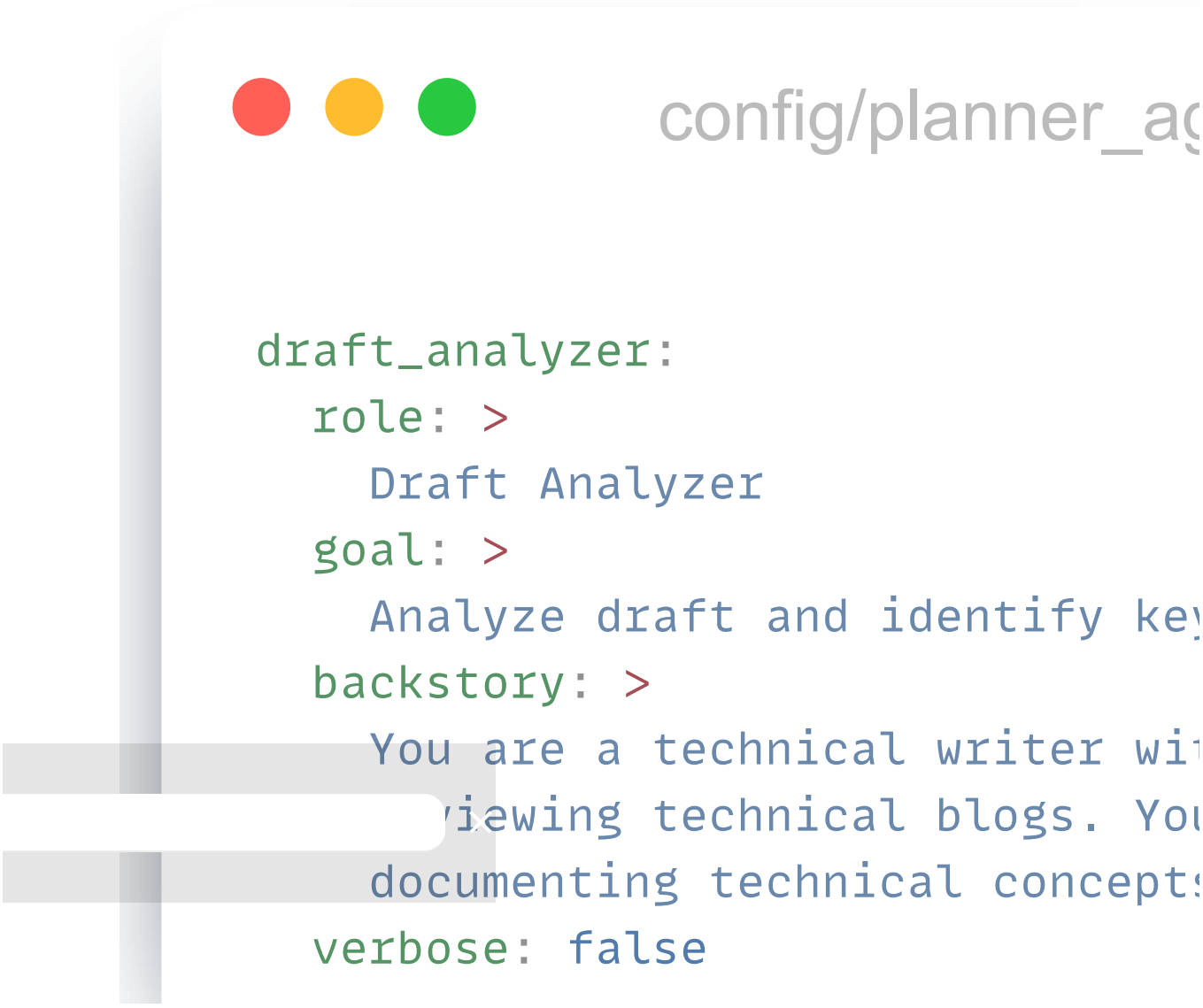
As we saw in previous parts, CrewAI works by defining Agents (AI workers) and Tasks (what they do).

In this particular setup, we need three agents, each specializing in a different part of the content planning workflow:

- Draft Analyzer → Extracts key ideas, sections, and media from the blog post.
- Twitter Thread Planner → Converts the blog content into an engaging Twitter thread.
- LinkedIn Post Planner → Creates a well-structured LinkedIn post optimized for professional

To build them, we first create a `config/` folder with two files:

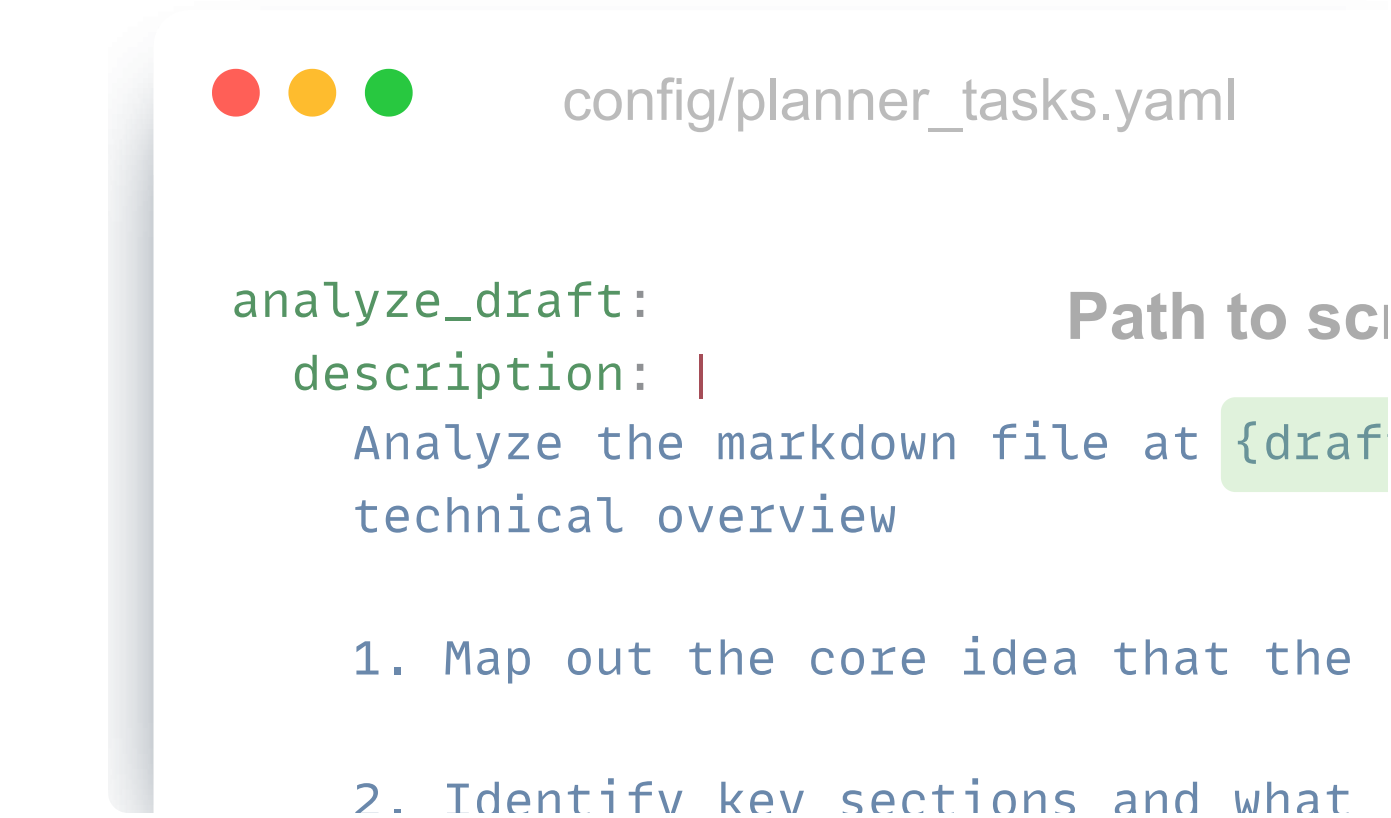
- `planner_agents.yaml` → Defines each agent's role, goal, and expertise.



- In the above code, we also specify the path to the scraped draft (which we shall scrape shortly) as a parameter— `{draft_path}` .

Moving on, each task in CrewAI is associated with a specific Agent.

- `planner_tasks.yaml` → Specifies what each agent is responsible for.
  - The Draft Analyzer extracts key sections, takeaways, and media references from the blog post:

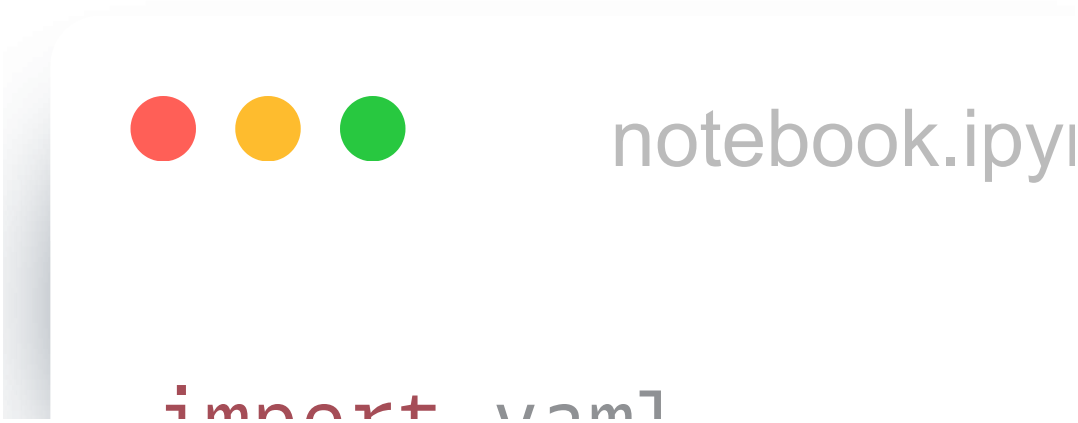


- The Twitter Thread Planner turns the analyzed blog into a structured Twitter thread

        - In the above config, we also specify the location of reference threads in the task description as a parameter `{path_to_example_threads}` for inspiration so that the LLM can match the writing style.
    - The LinkedIn Post Planner creates a long-form LinkedIn post optimized for professional audiences.

- In the above config, we also specify the location of reference threads in the task description as a parameter `{path_to_example_linkedin}` for inspiration so that the LLM can match the writing style.

Now that we have defined agents and tasks in YAML, we load them into our Python code:



Here's an important detail before we move onto defining the agents and their tasks.

The output structure can be unpredictable at times.

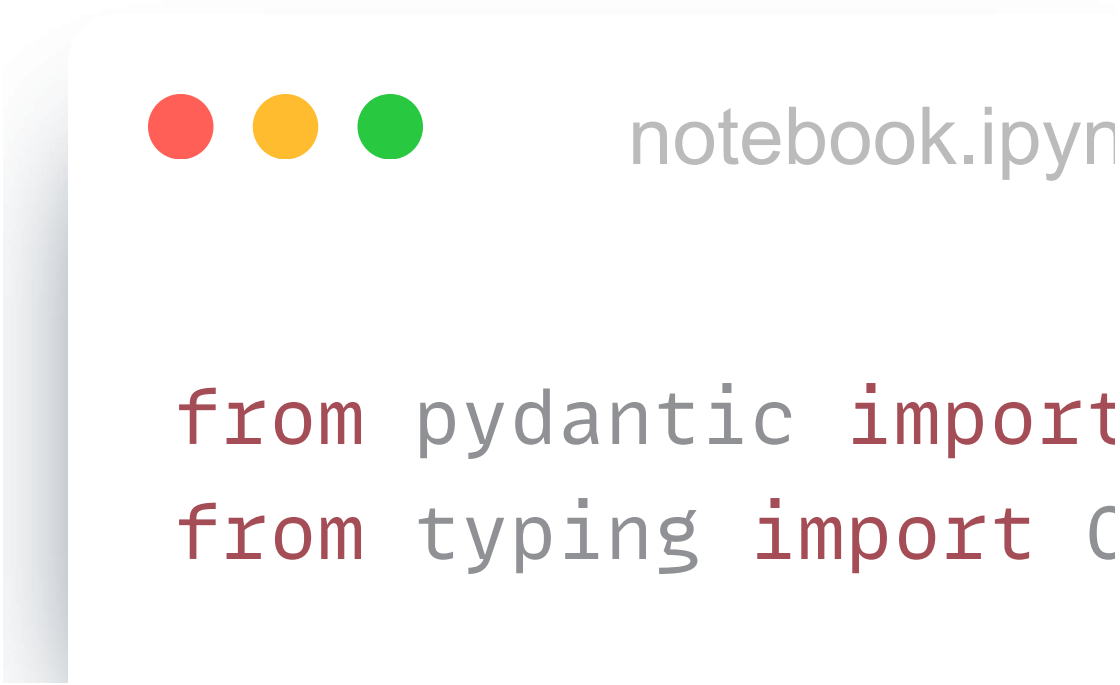
That is why it would be good that each CrewAI agent is responsible for generating structured outputs that adhere to specific formats.

So to ensure consistency and clarity, lets define structured models using Pydantic.

First, let's define the structured output for twitter thread planner. On Twitter:

- Each thread has a multiple tweets.
- And each thread has a text + visual/image (optional).

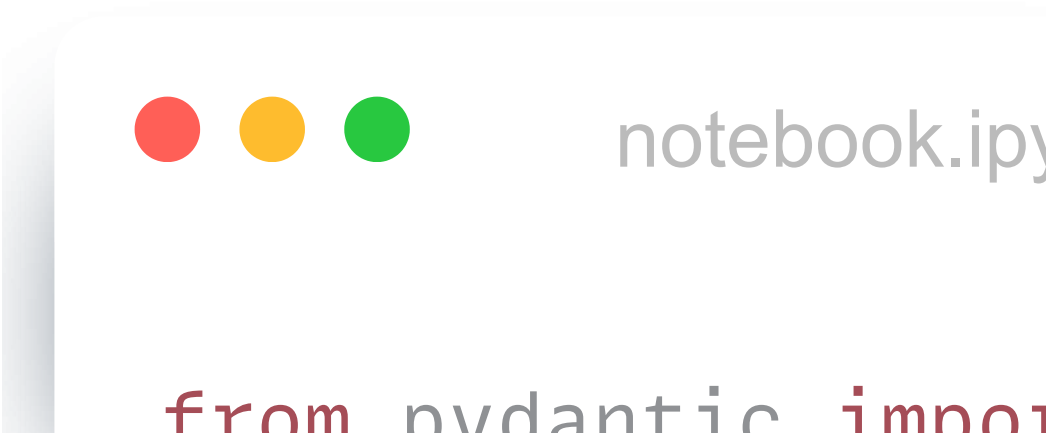
Thus, we define our structured output as follows:



In the above code:

- Planner agent generates a structured output based on the `Thread` Pydantic model.
- Each tweet in the thread follows the `Tweet` model, ensuring:
  - A hook tweet is explicitly marked (`is_hook=True`). It is the first tweet of the thread.
  - Media assets (images/code snippets) are associated correctly with each tweet.

Next, we define our structured output for LinkedIn.



- The LinkedIn Post Planner agent generates content adhering to this format.
- The `content` field ensures structured, well-formatted LinkedIn text.
- The `media_url` ensures an appropriate image is selected to enhance engagement.

The LinkedIn post is structured differently from a Twitter thread, prioritizing:

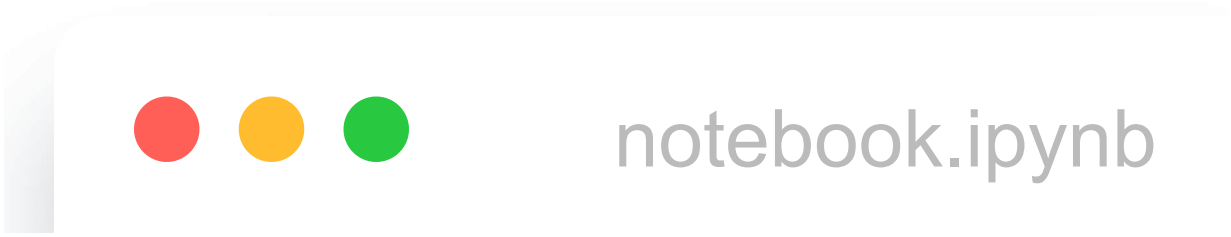
- [illegible] and in-depth analysis.
- Professional tone and engagement hooks.
- Strategic use of hashtags and calls to action.

Now that we understand the structured outputs, let's map the agents to these formats.

Each agent has to read some files for processing:

- Draft Analyzer Agent must analyze the scraped draft.
- Twitter Writer Agent must read the sample Twitter thread.
- LinkedIn Writer Agent must read the sample LinkedIn post.

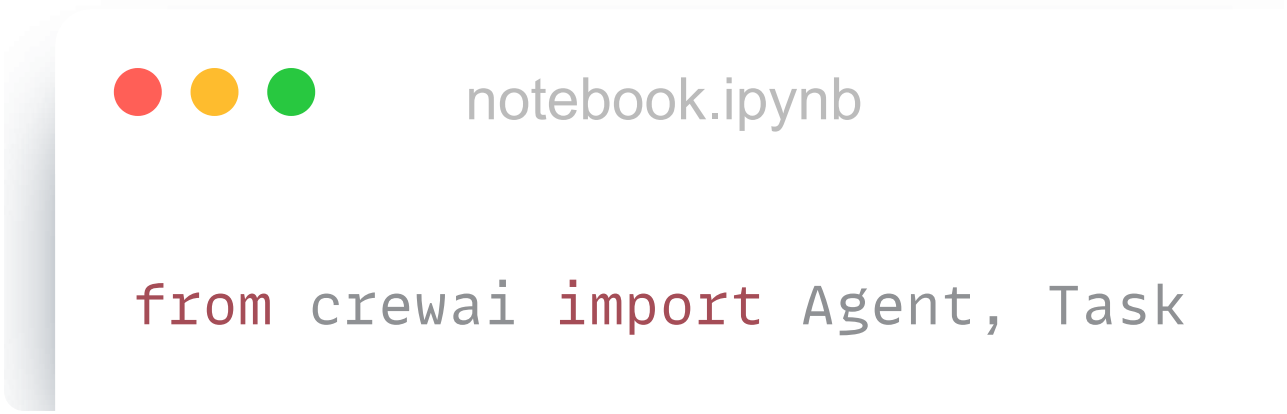
To facilitate this, we define two tools from CrewAI:



Now first, we have the Draft Analyzer Agent:

- Purpose: Extract key takeaways and media references from the blog post.
- Output: No structured output (acts as preprocessing).
- Usage: Feeds extracted data into both the Twitter and LinkedIn planners.

Next, we have the Twitter Thread Planner Agent:



- Purpose: Convert the blog post into a structured Twitter thread.
- Output Format: `Thread` (structured list of tweets).

Finally, we have the LinkedIn Post Planner Agent:

- Purpose: Generate a LinkedIn post optimized for professionals.
- Output Format: `LinkedInPost` (structured long-form content).

Build the Flow

Now that our agents and structured outputs are in place, we can move on to defining our Flow—the backbone of the content planning system.

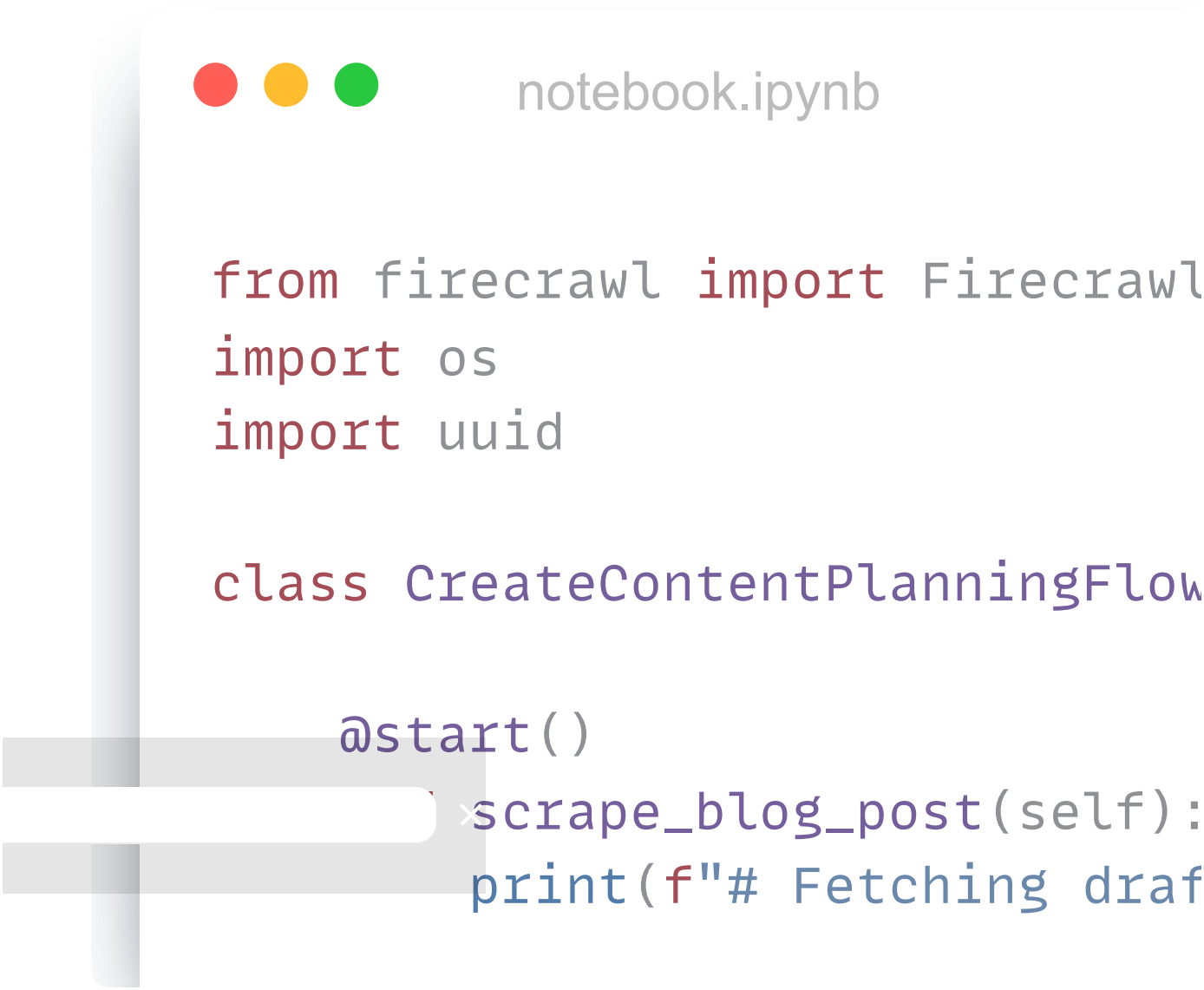
This CrewAI Flow orchestrates the entire workflow by:

1. Scraping the blog post using FireCrawl.
2. Selecting the target platform (Twitter or LinkedIn).

3. Generating structured content using AI agents.
4. Saving the final content plan for future use.

In CrewAI Flows, the state stores persistent values that are needed across different steps. Instead of using an unstructured dictionary, we define a structured state model using Pydantic as follows:

Moving on, the first step in our flow is to fetch the blog content using FireCrawl:



- FireCrawl scrapes the blog post and converts it into Markdown and HTML.
- The title is extracted from metadata (or a fallback ID is used).
- The content is saved locally at `assets/{title}.md` for later processing.

Once the blog is scraped, we need to determine whether to generate a Twitter thread or a LinkedIn post.

To do this, we use the `router` decorator:

`itter"`, the flow proceeds to generate a Twitter thread.

- If `post_type == "linkedin"`, the flow creates a LinkedIn post.

This ensures that only one of the content generation flows executes based on the user's selection.

Now we shall have two methods that will "listen" to the output of the above router method.

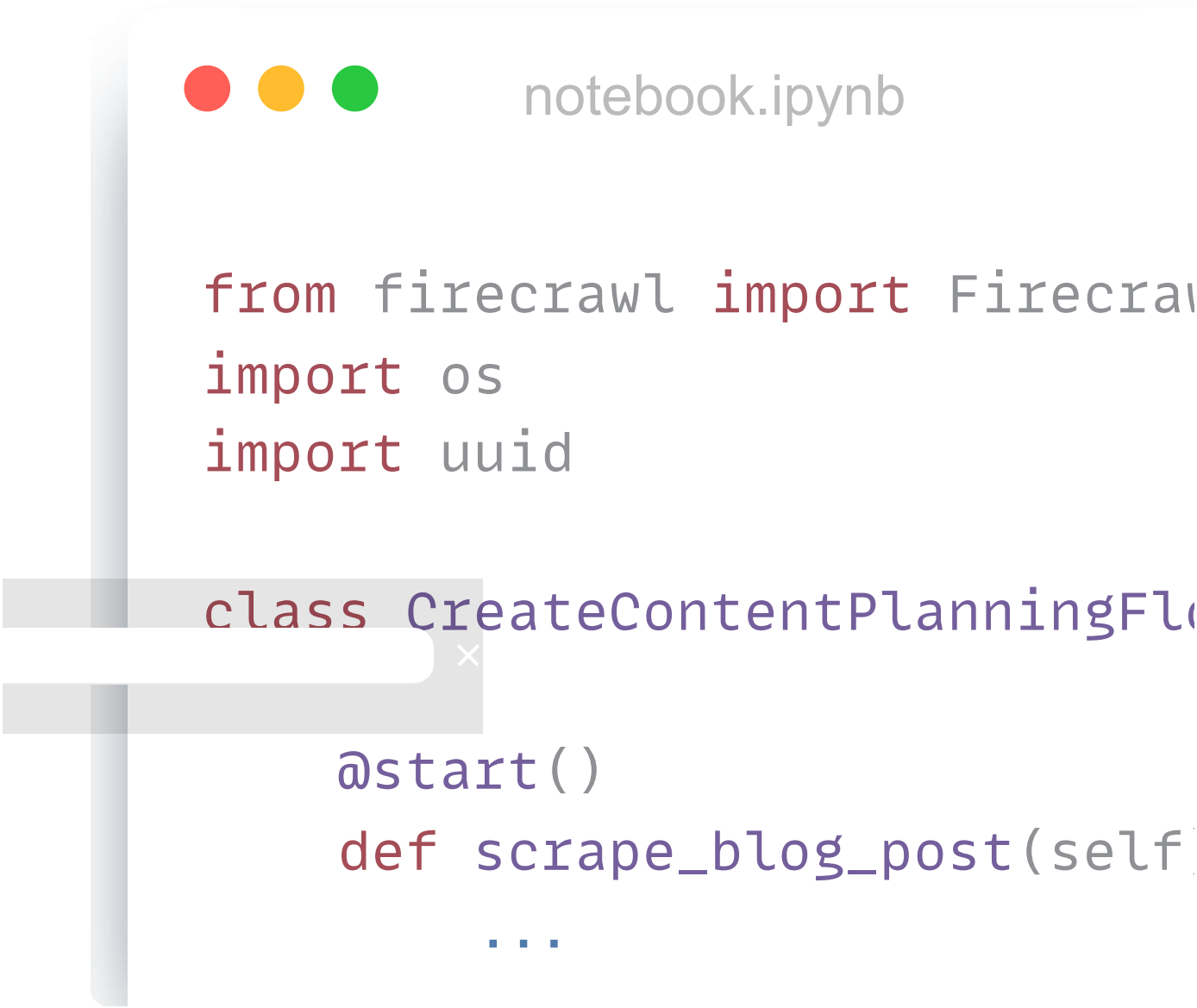
If the user has selected Twitter, the `twitter_draft` method will execute, which has been decorated with this decorator—`@listen("twitter")`:

The Twitter Planning Crew accepts the blog draft and generates a structured Thread output.

Each tweet is printed, showing:

- The content of the tweet.
- The associated media URLs (if any).

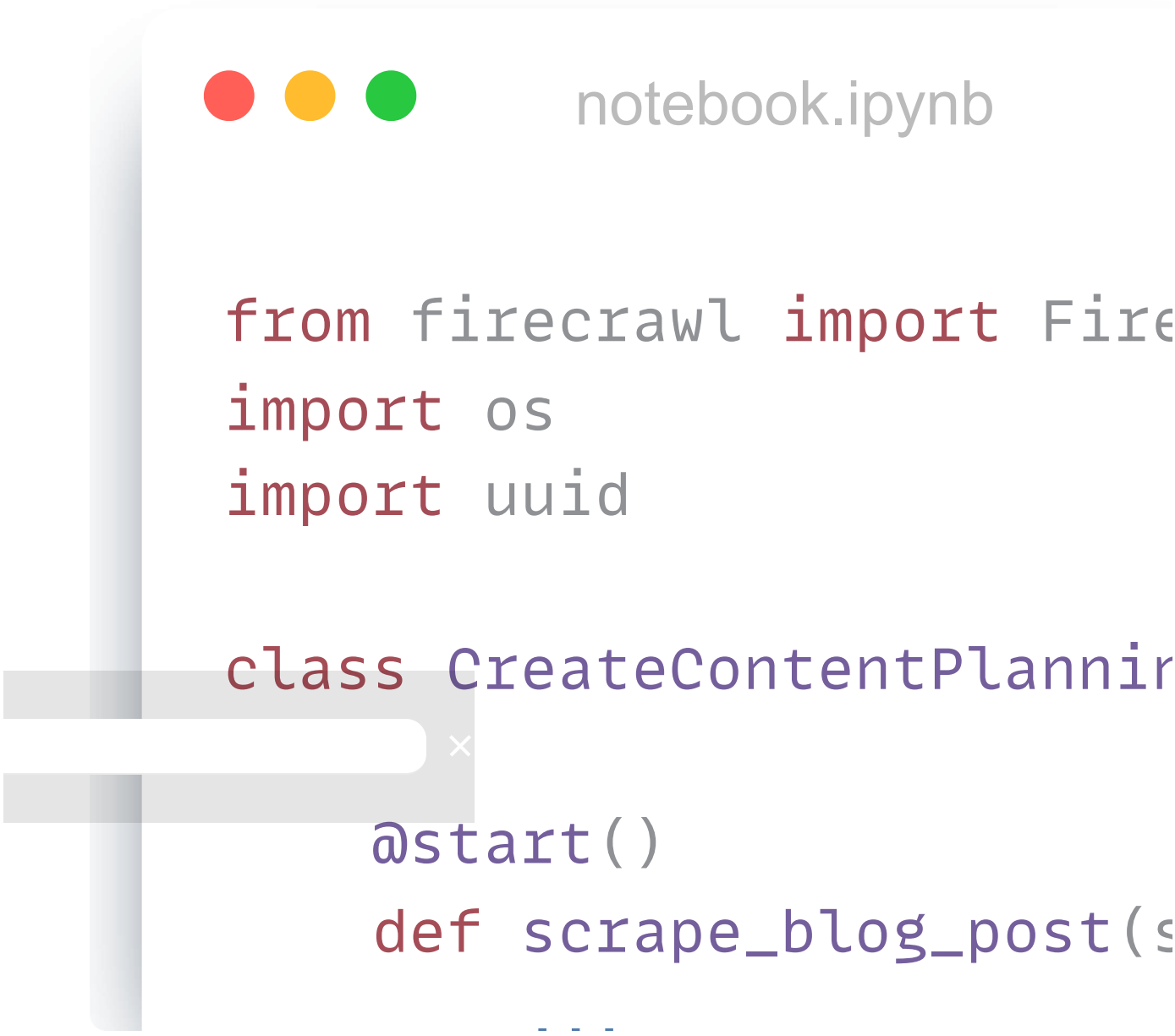
If the user has selected LinkedIn, the `linkedin_draft` method will execute, , which has been decorated with this decorator—`@listen("linkedin")`:



- The LinkedIn Planning Crew processes the blog and generates a structured `LinkedInPost` output.
- The generated LinkedIn post is printed for review.

Once the content is generated, the plan is saved as a JSON file for later use.

To do this, we make use of the listen decorator along with a conditional OR logic, which means if either linkedin method or twitter method finishes execution, the save method should run. This is implemented below:



Done!

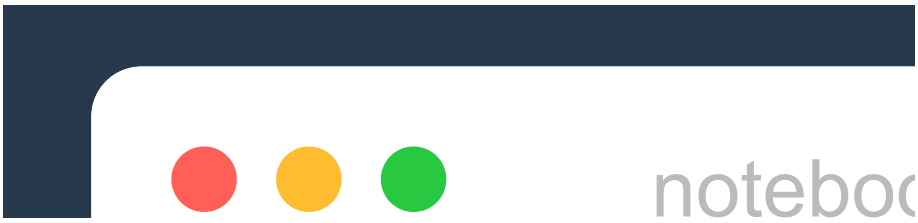
This CrewAI Flow automates the entire content repurposing pipeline:

- Scrapes blog content dynamically using FireCrawl.
- Chooses the correct platform (Twitter or LinkedIn) using routing logic.
- Generates structured social media content with AI-driven agents.
- Saves the content plan for future use or manual review.

Run the Flow

Now that we have defined our CrewAI Flow, it's time to execute it and see how our agents work together to generate structured content for social media.

To begin, we instantiate the `CreateContentPlanningFlow` class:



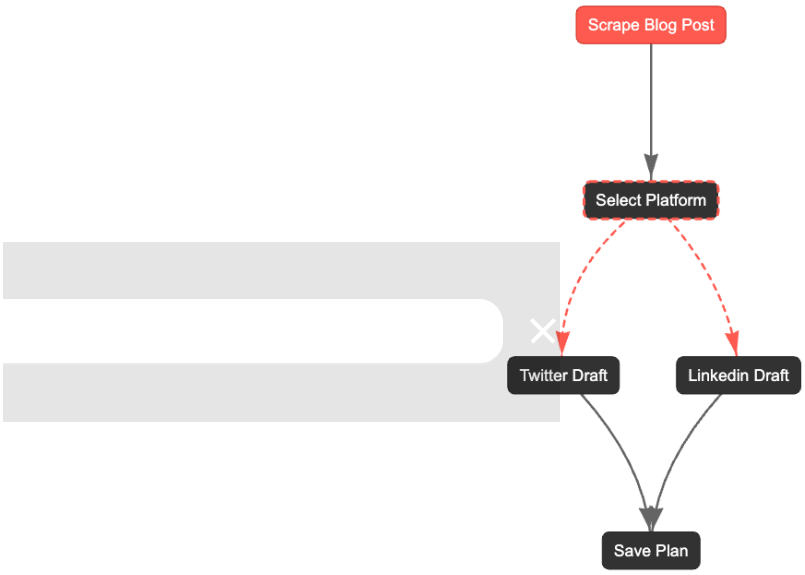
Here, the structured state (`ContentPlanningState`) is automatically assigned.

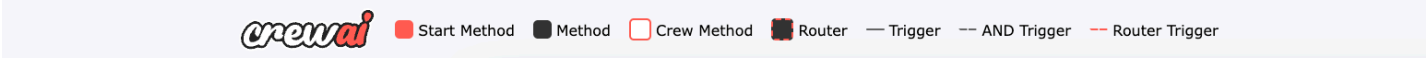
tate values are:

Before executing the flow, we can generate a visual plot to understand the execution sequence:

CrewAI generates a flow diagram and saves it as `crewai_flow.html`.

This visualization shows how each task connects within the workflow.





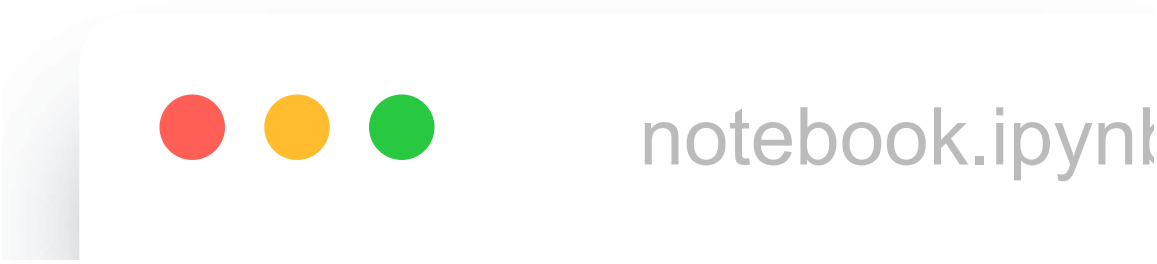
Now, we can execute the Flow using:

- FireCrawl scrapes the blog post and saves it in `assets/{title}.md`.
- The `@router` function chooses the correct Crew based on `post_type`.
  - If `"twitter"` → Twitter Planning Crew is executed.
  - If `"linkedin"` → LinkedIn Planning Crew is executed.
- The selected Crew:
  - Analyzes the blog draft (Draft Analyzer).
  - Generates a structured content plan (Twitter Thread or LinkedIn Post).
- The final content plan is saved as a structured JSON file in the `output/` directory.

This produces the following output and as you can see, we have the exact format as specified in the Pydantic model.

```
flow.kickoff()
✓ 29.2s

Flow started with ID: 90b27ba0-faad-41ec-9cf6-7999638968ef
# fetching draft from: https://blog.dailydoseofds.com/p/5-chunking-strategies-for-rag
# Planning content for: assets/5 Chunking Strategies For RAG - by Avi Chawla.md
# Planned content for assets/5 Chunking Strategies For RAG - by Avi Chawla.md:
Tweet 1:
5 Chunking Strategies For RAG
Media URLs: []
--------------------------------------------------------------------------------
Tweet 2:
Chunking is crucial in RAG systems to efficiently process large documents. 📚
Here's how it fits into the RAG workflow with data stored as vectors, matched for queries, and generating responses:
Media URLs: ['https://substack-post-media.s3.amazonaws.com/public/images/6878b8fa-5e74-45a1-9a89-5aab92889126_2366x990.gif']
--------------------------------------------------------------------------------
Tweet 3:
Let's break down the 5 essential chunking strategies, starting with **Fixed-size Chunking**.

It's easy to implement but might disrupt the semantic flow. 🔪
Media URLs: ['https://substack-post-media.s3.amazonaws.com/public/images/98c422a0-f0e2-457c-a256-4476a56a601f_943x232.png']
--------------------------------------------------------------------------------
Tweet 4:
**Semantic Chunking** segments text based on meaning using cosine similarity. 🤔

This helps in maintaining idea integrity, but requires a threshold setting.
Media URLs: ['https://substack-post-media.s3.amazonaws.com/public/images/74037e11-362d-4ea2-8ee2-ee85ab013523_963x231.png']
--------------------------------------------------------------------------------
Tweet 5:
**Recursive Chunking** uses natural separators and divides large sections.🔍

This approach preserves semantic integrity while adding complexity.
Media URLs: ['https://substack-post-media.s3.amazonaws.com/public/images/b0e40cc1-996f-48f4-9306-781b112536e4_984x428.png']
--------------------------------------------------------------------------------
Tweet 6:
**Document Structure-based Chunking** utilizes existing document structure to set boundaries. 🏛

While chunk sizes might vary, this method preserves structural integrity.
Media URLs: ['https://substack-post-media.s3.amazonaws.com/public/images/40bdaf3b-601d-4357-bc7f-89b47f812097_1025x663.png']
--------------------------------------------------------------------------------
Tweet 7:
Lastly, **LLM-based Chunking** brings semantic accuracy using LLMs. 🤖

Despite being computationally demanding, it ensures better semantic coherence.
Media URLs: []
--------------------------------------------------------------------------------
Tweet 8:
Chunking optimizes response quality in RAG applications, balancing complexity with performance.

Semantic & LLM chunking provide better coherence but at a higher cost.
Media URLs: []
--------------------------------------------------------------------------------
Tweet 9:
Understanding these strategies allows developers to choose the best approach based on resources and desired outcomes for RAG applications.
Media URLs: []
--------------------------------------------------------------------------------
Tweet 10:
That's a wrap on chunking strategies in RAG!

Follow for more insights on AI and Engineering!
Media URLs: []
--------------------------------------------------------------------------------
```

Similarly, we can also execute the LinkedIn flow as follows:



Simple, isn't it?

Demo 2: Book Writer Flow

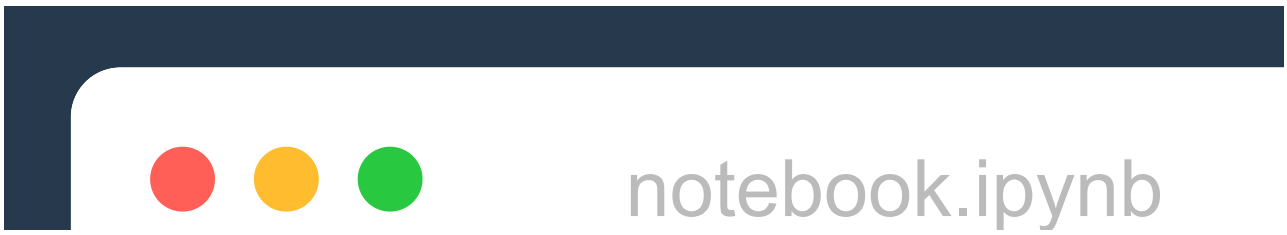
The discussion we had above was our first ever multi-crew Flow workflow.

CrewAI Flows supports multiple crews which allows us to build specific Crews, each [illegible]bilities.

Once done, we can orchestrate them in a single flow.

As we saw in previous part, the CrewAI framework makes it seamless to integrate Crews into Flows.

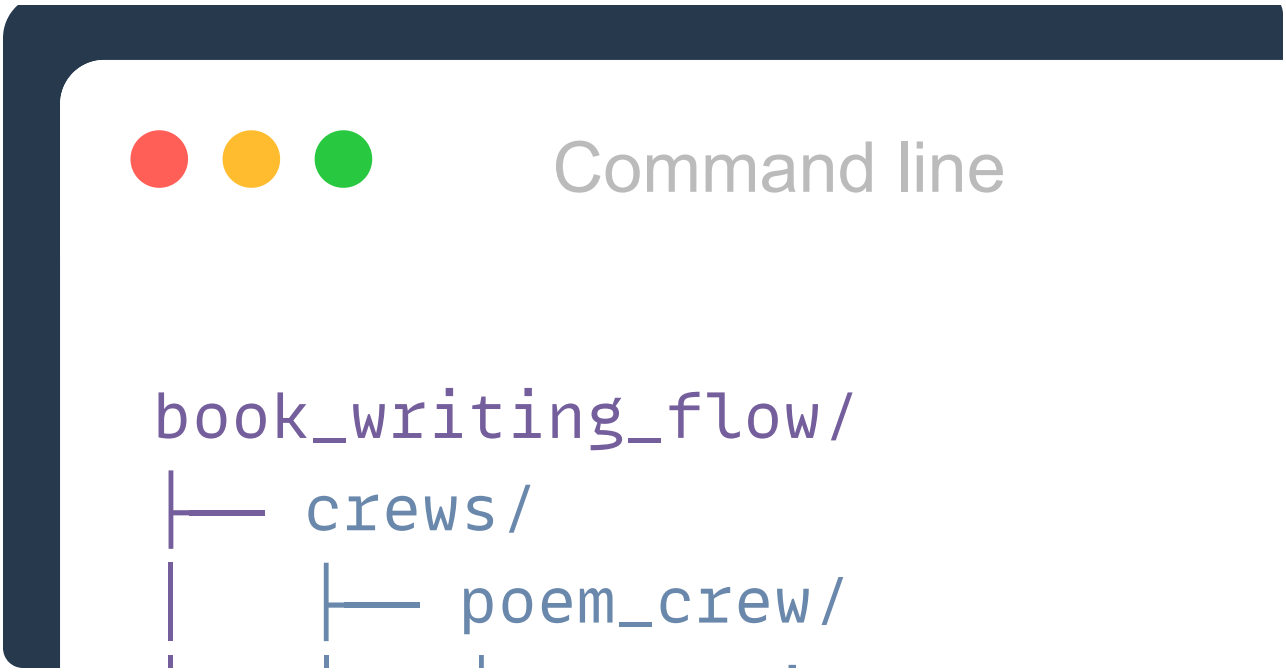
To create a new Crew-enabled Flow in CrewAI, create a new directory using either of the two ways below:



Also, move to the above directory before proceeding ahead.

Next, run the following command to create a Flow project:

As discussed last time, this will automatically generate a project with all necessary files and directories, and the project structure will look like this:



The project is structured into separate components of the workflow. Each part has a dedicated role, making it easy to scale and maintain. Below is a breakdown of the directory structure for `book_writing_flow/` and its purpose:

- `crews/`:
  - This folder holds all Crews (groups of AI agents working together). Each Crew must have its own directory.
  - `poem_crew/` → A specific Crew (e.g., for generating poetry).
    - `config/` → Stores configuration files for this Crew.
      - `agents.yaml` → Defines the agents involved like we learned in Part 2.
      - `tasks.yaml` → Defines the tasks assigned to agents.
    - `poem_crew.py` → Python script that defines the Crew by loading the agents/tasks YAML file, loading the tools, etc.
  - To add another Crew, you must add another folder to this directory. Alternatively, you can copy and paste the above `poem_crew` folder, rename it, and then modify the YAML files and the Crew script.
  - contains custom tools that AI agents can use.
  - `custom_tool.py` → A script where you can define custom functions (e.g., fetching data from APIs, formatting text, performing calculations).
- `main.py`: This is the entry point of the project. It orchestrates the flow, connecting different Crews and defining how tasks are executed.
- Rest, we have some self-explanatory files like README, gitignore, etc.

Below, let's extend this Flow to a multi-crew setup and create a Book Writer Flow.

But first, let's look at how we will go about building this Flow—how many Crews would be needed, which tools would be needed, etc.

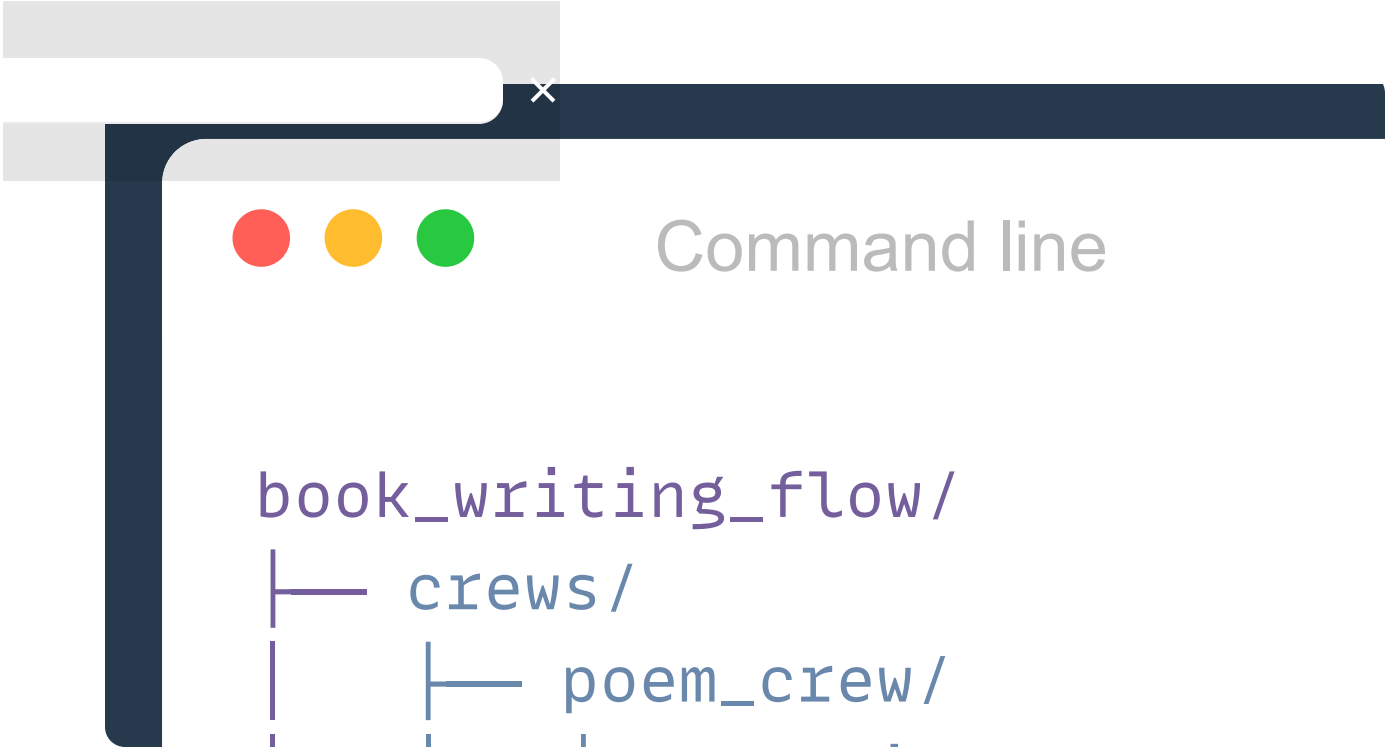
- First, we need a Outline Crew that will have two agents:
  - Researcher Agent → Accepts the topic, searches the web for it, and gather insights.
  - Outline Writer Agent → Takes the insights from the Researcher Agent, decides the number of chapter to write in the book and the title of each

chapter.

- Next, we need a Chapter Writer Crew that will have two agents:
  - Researcher Agent → Accepts the title of a chapter, searches the web for it, and gather insights.
  - Chapter Writer Agent → Takes the insights from the Researcher Agent of this Crew and actually writes that chapter for us.
- If the Outline Crew suggests 10 chapters, for instance, then we must run 10 Chapter Writer Crews in parallel, where each Crew will write one chapter each.
- Whenever we need to search the web, we can use the Serper Dev tool.

Now that we have outlined the process, let's move on to implementing the Book Writer Flow.

Consider the current project structure:



To begin, we need to make these three changes in this directory

- We make a copy of the `poem_crew` folder in the `crews/` folder.
- We rename `poem_crew` to `Outline_crew` and set the name of copied folder to `Writer_crew`.

- We rename `Outline_crew/poem_crew.py` file to `outline_crew.py` and the `Writer_crew/poem_crew.py` file to `writer_crew.py`.

Finally, the project structure should look like this:

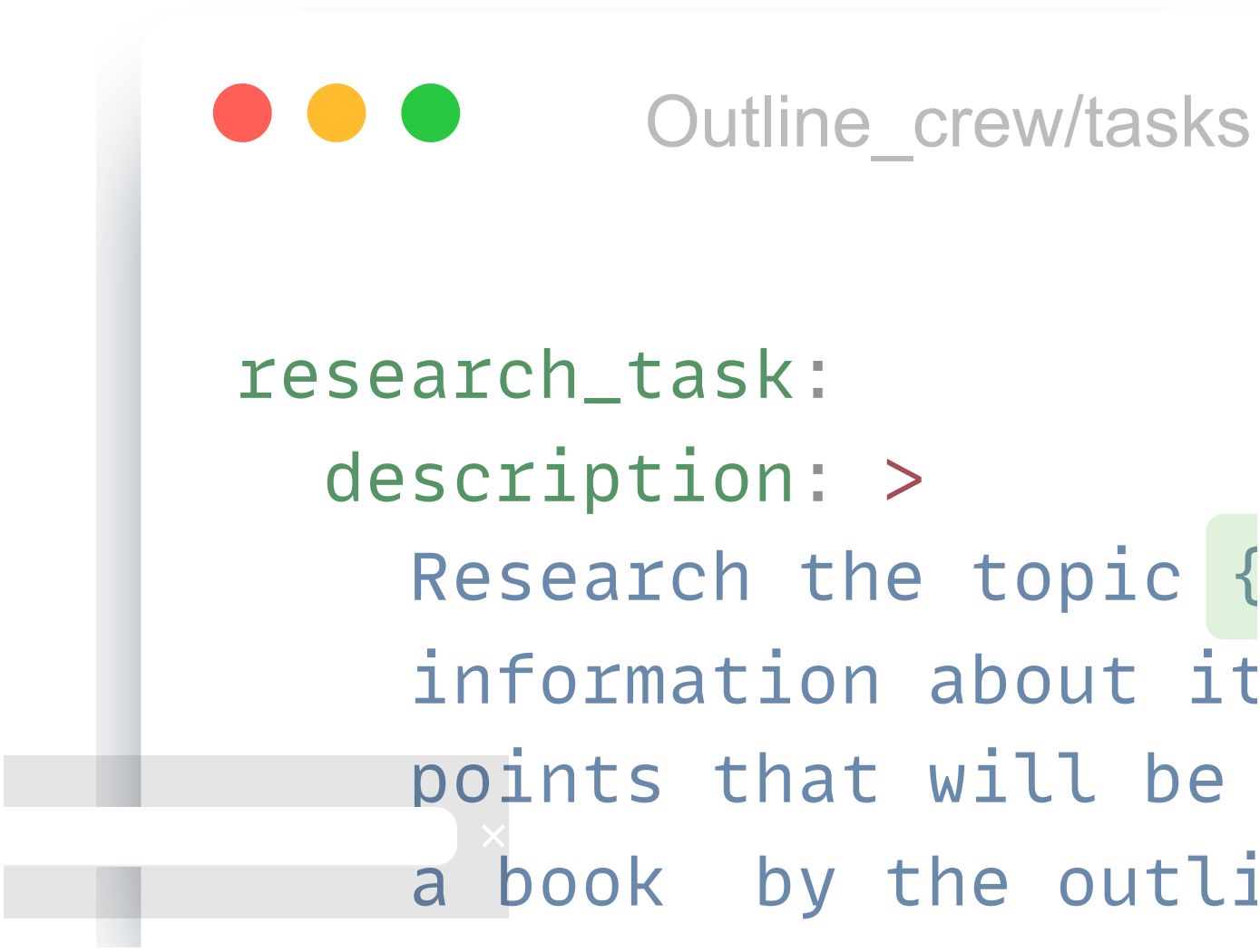
Now let's move onto updating the two Crew folders, their config files, Agents, Tasks, and Tools.

First, let's make changes to the `Outline_crew/agents.yaml` file as follows:

As discussed earlier, we have two agents—a research agent that gathers insights about the topic and an outline agent that will decide the number of chapters and chapter titles.

In the above YAML config, we have specified the `{topic}` as a parameter that will be passed during Flow execution. We'll get to it shortly but for now, just assume that there exists a `{topic}` parameter.

Now, we move onto defining the Tasks for these two Agents as follows:



In the above config file, we have defined two Tasks for our Agents—one for each Agent. The task for the `outline_writer` agent expects it to output the total number of chapters in the book and the list of titles of each chapter.

But just specifying expected output in the structure is not sufficient. We need to create a structured output for this so let's do that below in the `Outline_crew/outline_crew.py` file.

First, we define our structured output for the Outline Agent:

Next, we orchestrate our entire Crew just like we learned in Part 2 of this Crash Course:

In the above code, we:

- Define the YAML config files.
- Define the two Agents and their tasks.

    - The Researcher Agent has access to the Serper Dev tool.
    - The Task of the Outline Agent expects a Pydantic output.
- Define the Crew.

Done!

> 💡 Note: We will be using OpenAI models for this Flow so if you don't have an OpenAI API key, we recommend getting one. If you want to do it locally through Ollama, you can do that too. In the above outline_crew.py file, create an `llm` object (just like we learned earlier) and pass it as the `llm=` parameter to the two Agents. This will work.

Now, let's move to the next Crew—the Writer Crew which will accept the output of the Outline Crew and draft the book for us. This Crew will have two agents:

- Researcher Agent → Accepts the title of a chapter, searches the web for it, [illegible]ghts.
- Chapter Writer Agent → Takes the insights from the Researcher Agent of this Crew and actually writes that chapter for us.

First, let's look at the Agent YAML config file— `Writer_crew/agents.yaml` :

Notice in the above code that we have defined two Agents. Since a book will have many chapters, we shall have that many Crews running in parallel and each Crew will write one chapter. Thus, we have passed these parameters to the Agents:

- `{title}` → indicates the title of the chapter to be written by the Crew.
- `{topic}` → indicates the topic of the book so that the Agent knows about the overall theme.
- `{chapters}` → indicates the the list of chapter titles so that the Agent writing a particular chapter knows the overall global picture.

Next, let's look at the Tasks YAML config file—`Writer_crew/tasks.yaml`:

Next, we need to test our entire Chapter Writer Crew just like we learned in Part 2 of this Crash Course.

Ideally, it would be good if our Chapter Crew also follows a structured output, which is specified below:

Once that's done, we orchestrate our entire Crew as follows:

In the above code, we:

- Define the YAML config files.
- Define the two Agents and their tasks.
    - The Researcher Agent has access to the Serper Dev tool.

- The Task of the Chapter Writer Agent expects a Pydantic output.

With that, we have built our Crews.

Our next task is to build the Flow where we shall bring together these Crews and define the Flow class while using `start`, `listen`, etc., decorators we learned earlier.

To do this, go to the `main.py` file in the `book_writing_flow` folder marked below:

You should already see some code in this file so remove that.

The `main.py` file in `book_writing_flow/` defines the execution logic for the book-writing process. We begin by:

- Importing necessary components, including `Flow`, `start`, `listen`, and `asyncio` (for concurrent processing because we will launch multiple Crews in parallel to write chapters).
- Loading environment variables using `dotenv` (e.g., API keys for AI models).
- Import the two Crews created:
  - `OutlineCrew` → Generates the chapter outline.
  - `ChapterWriterCrew` → Writes content for each chapter.

Since we need a structured format for the book, we define Pydantic models for chapters and also for state management of our Flow.

We used this format because it:

- Ensures that each chapter has both a title and content.
- Helps structure the generated text into a consistent format.
- Makes it easier to store, edit, and save the book later.

Also, our state management easily:

- Tracks the book's topic (`"Astronomy in 2025"` by default).
- Stores the total number of chapters.
- Maintains a list of chapter titles (from the outline).
- Holds the written chapters in a structured format (`Chapter`).

The `BookState` ensures that data persists across multiple steps in the Flow.

With our structured models in place, we now define the CrewAI Flow that
cess.

- The `@start()` decorator ensures this step runs first.
- The OutlineCrew is executed to generate a structured book outline.
- Chapter count and their titles are stored in `self.state` defined earlier.

Once the outline is ready, we generate each chapter asynchronously to speed up processing.

This is implemented below:

main.py

```python
class ChapterFlow(Flow[ChapterState]):

    @start()
    def generate_outline(self):
        ...

    @listen(generate_outline)
    async def generate_chapters(self):
        print("Generating chapters")
        tasks = []

        async def write_single_chapter(title: str):
            result = (
                ChapterWriterCrew()
                .crew()
                .kickoff(inputs={
                    "title": title,
                    "topic": self.state.topic,
                    "chapters": [chapter.title for chapter in self.state.chapters]
                })
            )
            return result.pydantic

        # Create tasks for each chapter
        for i in range(self.state.total_chapters):
            task = asyncio.create_task(write_single_chapter(self.state.titles[i]))
            tasks.append(task)

        # Wait for all chapters to be generated concurrently
        chapters = await asyncio.gather(*tasks)
        print(f"Generated {len(chapters)} chapters")
        self.state.chapters.extend(chapters)
```

Runs after outline method

Run multiple crews in parallel

- `@listen(generate_outline)` → This step executes after the outline is created.
- `async def write_single_chapter(title: str)` → Defines an asynchronous task for writing each chapter.
  - Within this method, we invoke the ChapterWriterCrew with the inputs like—chapter title, topic and list of chapters.
- `asyncio.create_task()` → Creates a separate task for each chapter so multiple chapters can be written in parallel.

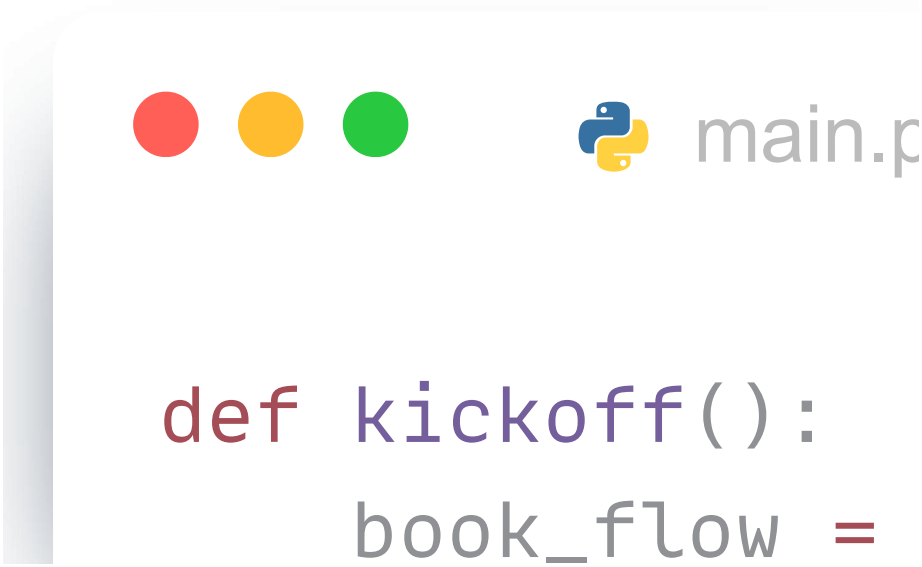
- `await asyncio.gather(*tasks)` → Waits for all chapters to finish writing before proceeding.

Writing multiple chapters in parallel makes execution much faster. Each chapter is written independently, preventing unnecessary dependencies.

Once all chapters are written, we save the book as a Markdown file.

- Each chapter's title and content are written to `book.md`.
- The Markdown format ensures the book is readable and easy to format later.

To execute the flow, we define two utility functions:



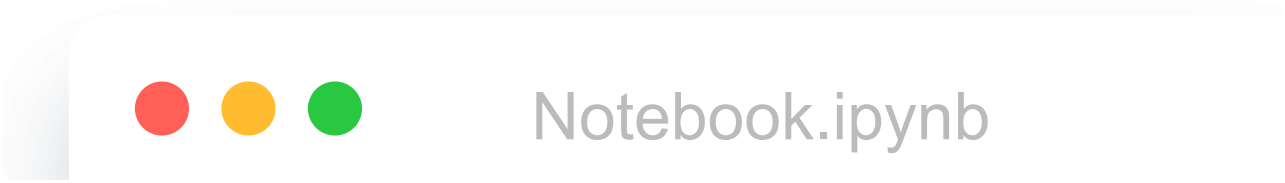
We have implemented our multi-crew Book writer Flow.

One final thing before we execute it. In the .env file marked below, add these two API keys:

You can get the Serper Dev API key here: https://serper.dev/.

Done!

Next, to execute it, move to the `src/` folder of your project using the command line or jupyter notebook:



Once done, run the Flow as follows:

And we get the following output when we run the Flow:

0:01 / 1:04

And we also get this Markdown file as the output:

0:01 / 0:14

Impressive, isn't it?

Conclusion, takeaways, and next steps

With that, we come to the end of Part 4 of the Agents crash course.

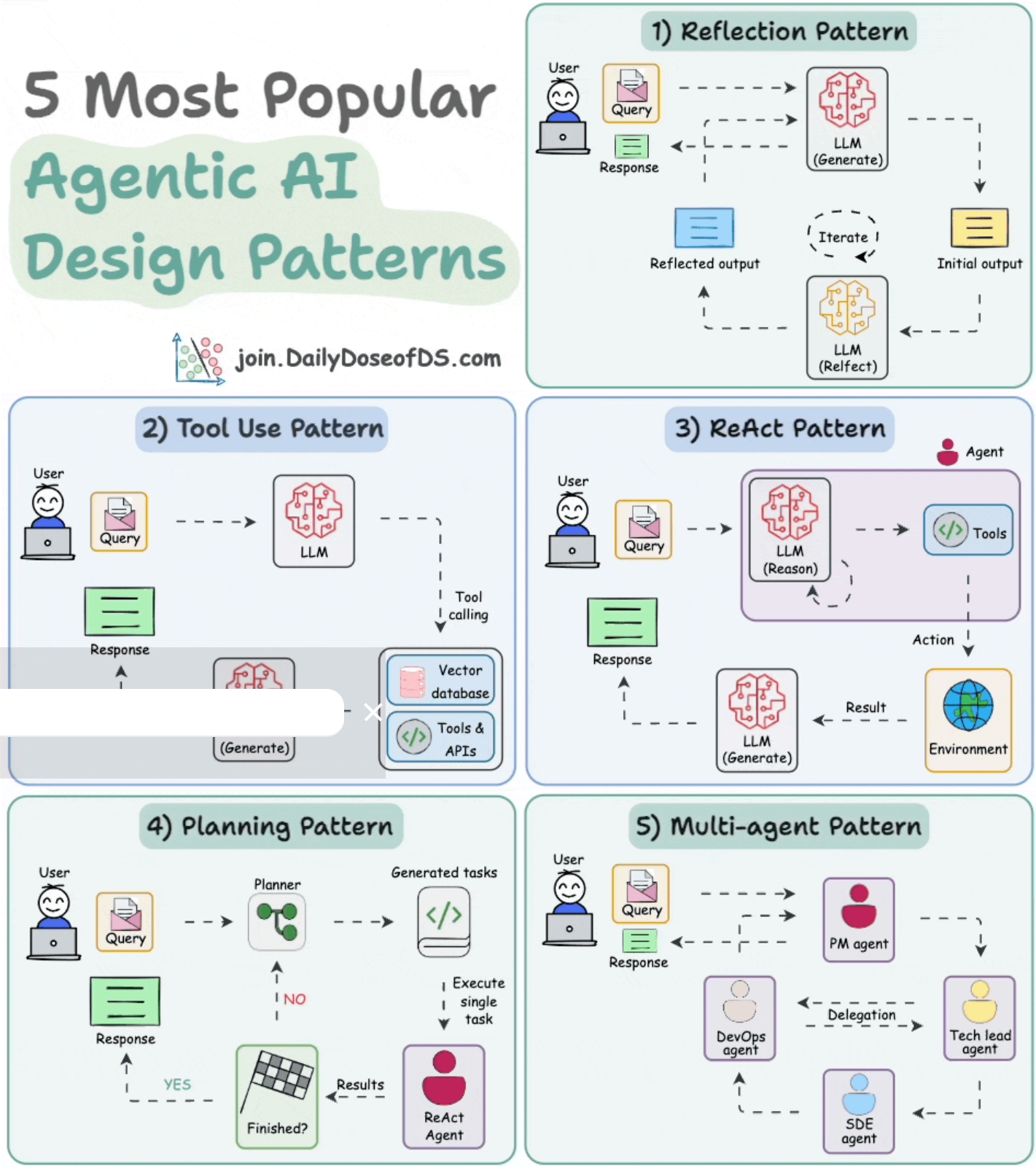
In this part, we learned how to build powerful multi-crew Flow systems with CrewAI, focusing on how to:

- Structure a multi-agent workflow → Defining specialized Crews, each handling different parts of a complex task.
- Use structured state management → Ensuring data persists across different execution steps in a Flow.
- Implement flow control mechanisms → Using `@start`, `@listen`, and `@router` to manage execution dependencies.
- Run asynchronous execution → Using `asyncio` to parallelize tasks and improve efficiency.
- Save structured AI-generated content → Writing results to Markdown or JSON for easy reuse.

As an exercise, try to implement the entire book writing flow yourself.

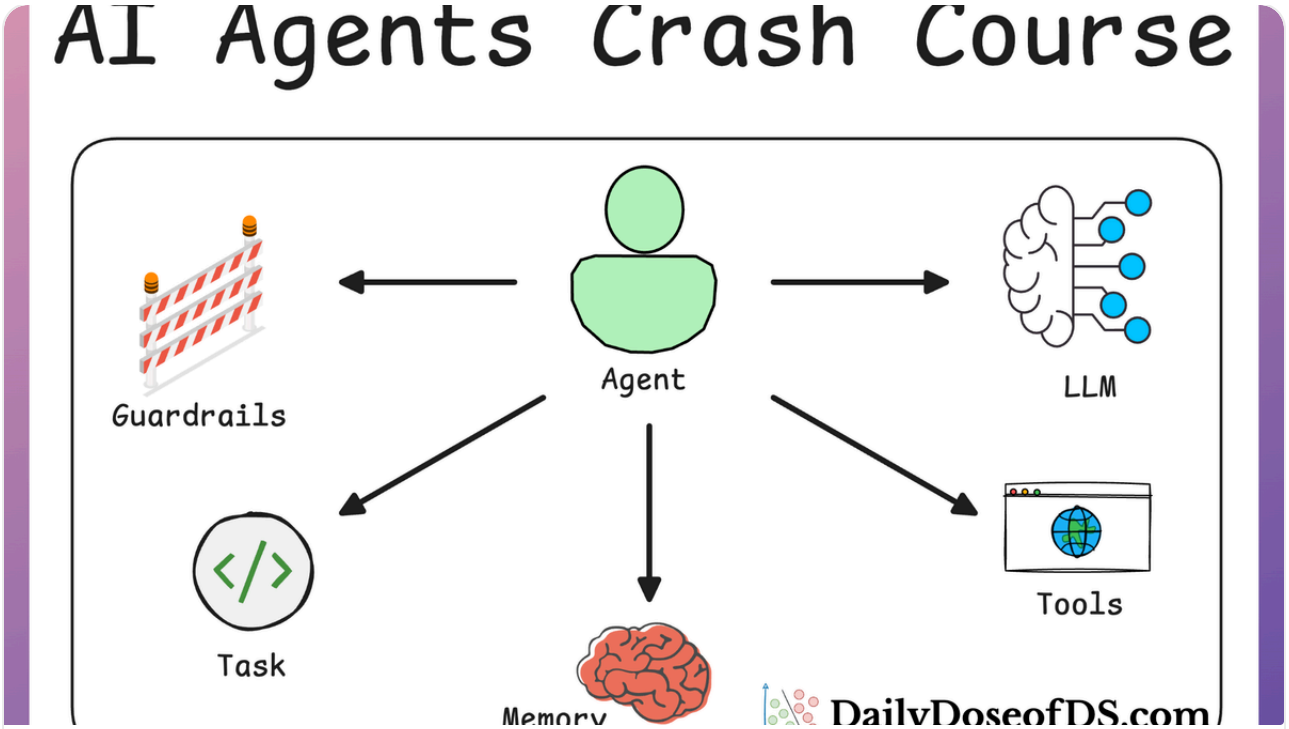
In the upcoming parts, we have several other advanced agentic things planned for you:

- Building production-ready agentic pipelines that scale.
- Creating and integrating custom tools for enhanced agent capabilities.
- 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 here:



Advanced Techniques to Build Robust Agentic Systems (Part A)

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

Daily Dose of Data Science • Avi Chawla

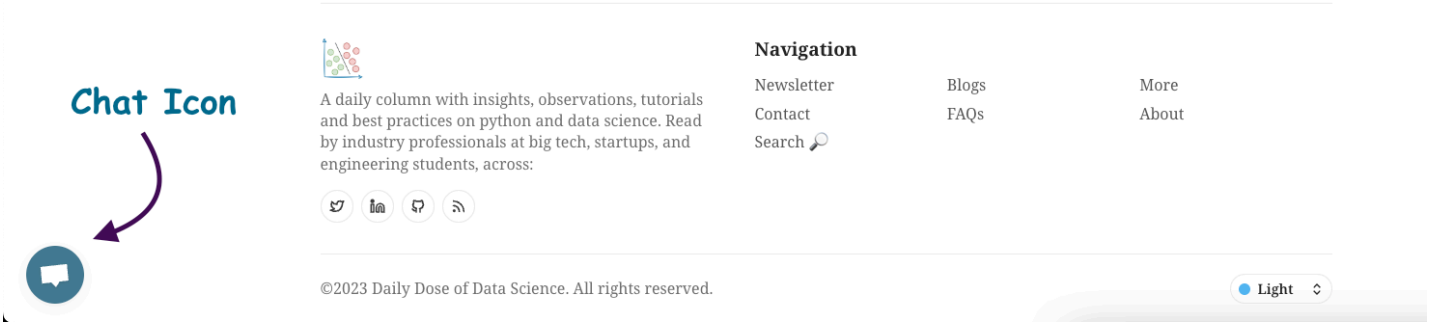
As always, thanks for reading!

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