Feb 23, 2025

Building Flows in Agentic Systems (Part A)

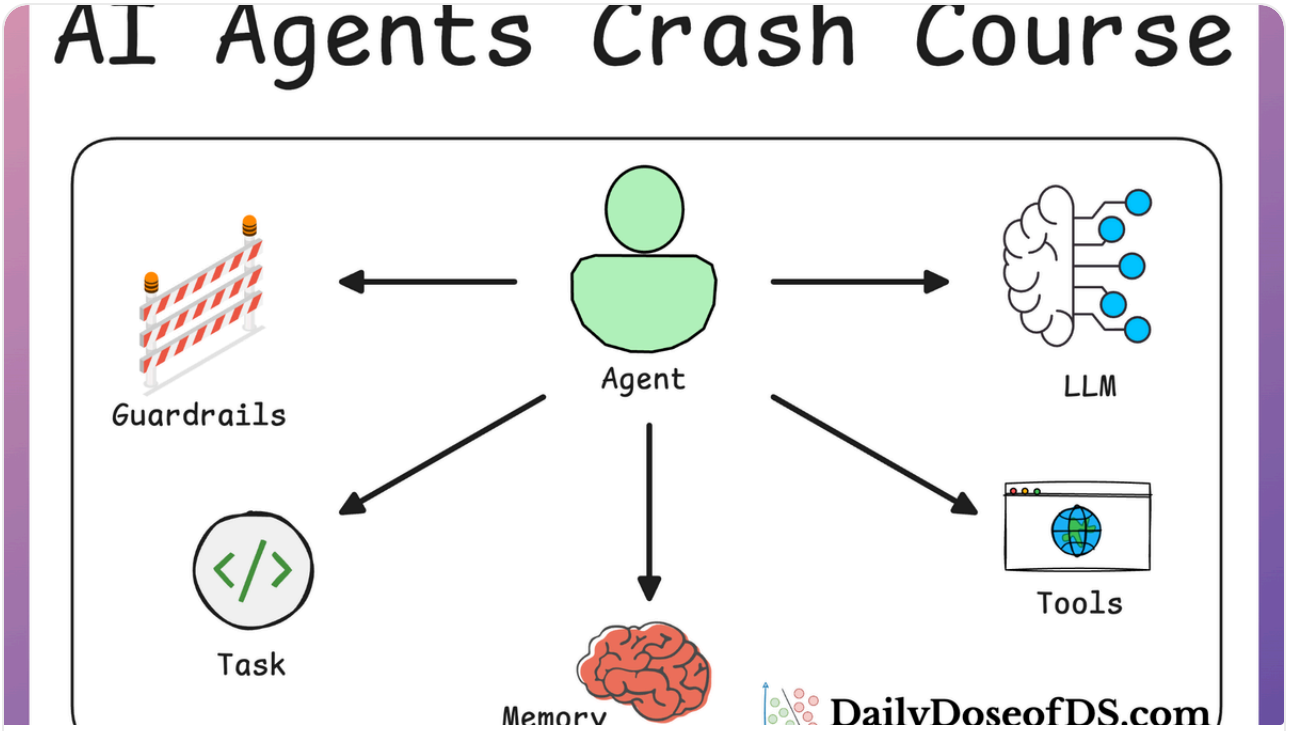
AI Agents Crash Course—Part 3 (with implementation).

Avi Chawla, Akshay Pachaar

Introduction

In the previous parts of this crash course, we explored the foundational elements of building agentic systems:

- In Part 1, we introduced the concept of AI agents, discussing their roles, goals, and how they can autonomously perform tasks.

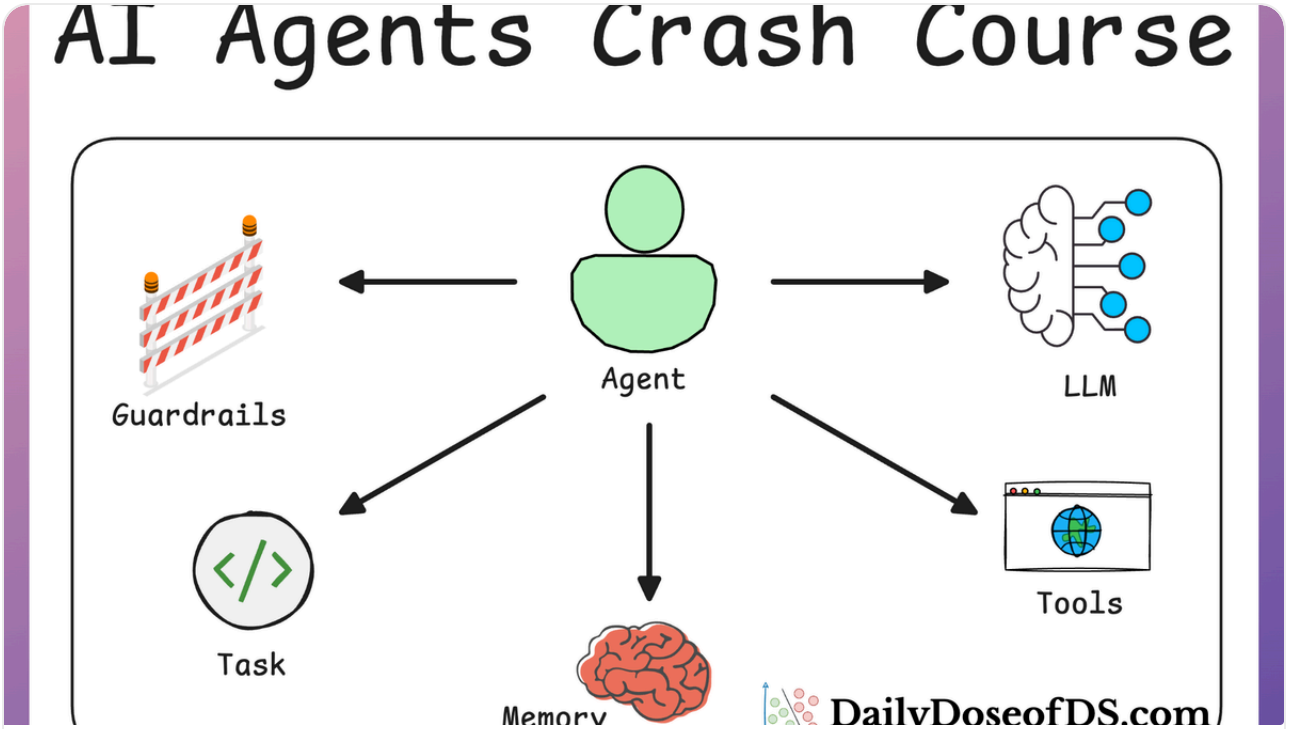


AI Agents Crash Course—Part 1 (With Implementation)

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 dived into structuring AI workflows by integrating custom tools for better multi-agent collaboration. This included building modular Crews and implementing structured outputs to enhance efficiency and scalability.



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

As we progress to Part 3, our focus shifts to CrewAI Flows—a powerful feature designed to streamline the creation and management of AI workflows.

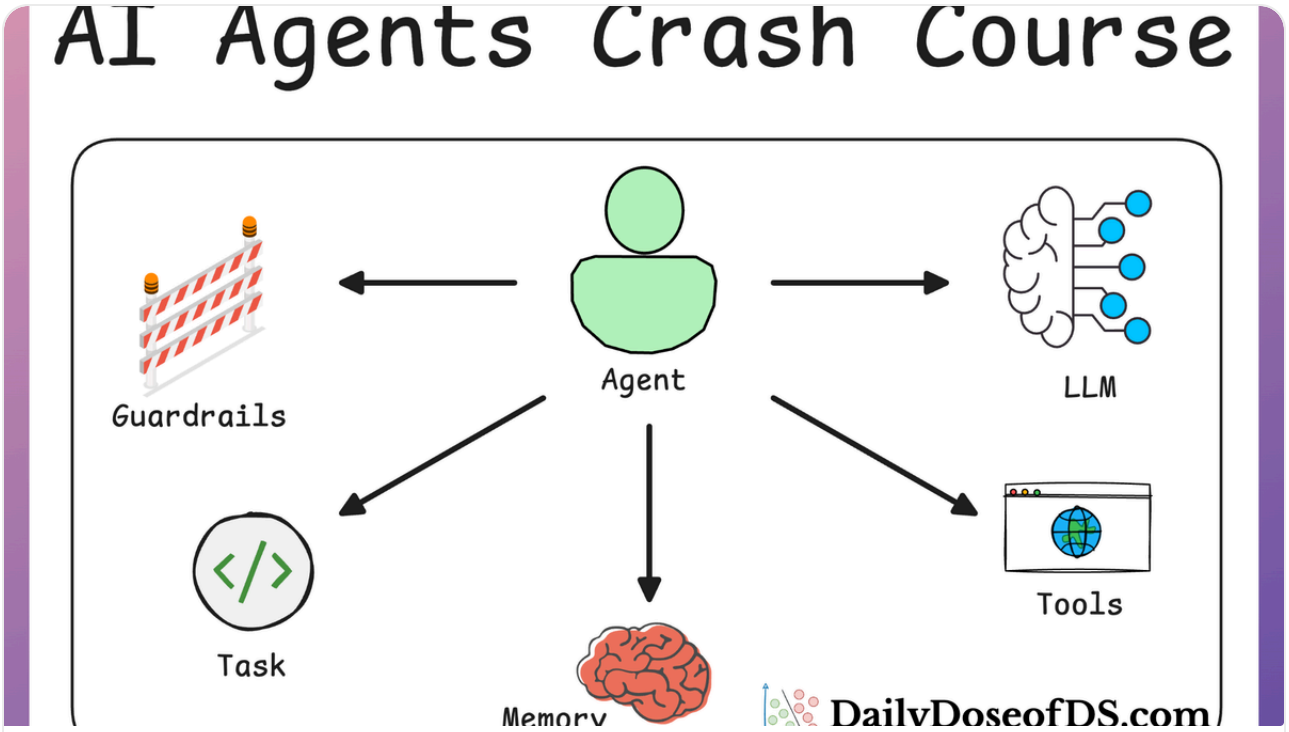
With Flows, you can create structured, event-driven workflows that seamlessly connect multiple tasks, manage state, and control the flow of execution in your AI applications. This allows for the design and implementation of multi-step processes that leverage the full potential of CrewAI's capabilities.

In this part, we will explore how we can use Flows to:

- Bring together multiple Crews and tasks to build complex AI workflows.
- Set up event-driven AI workflows.
- Manage states across the entire AI workflow.
- Incorporate conditional logic and branching to build flexible AI workflows.

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

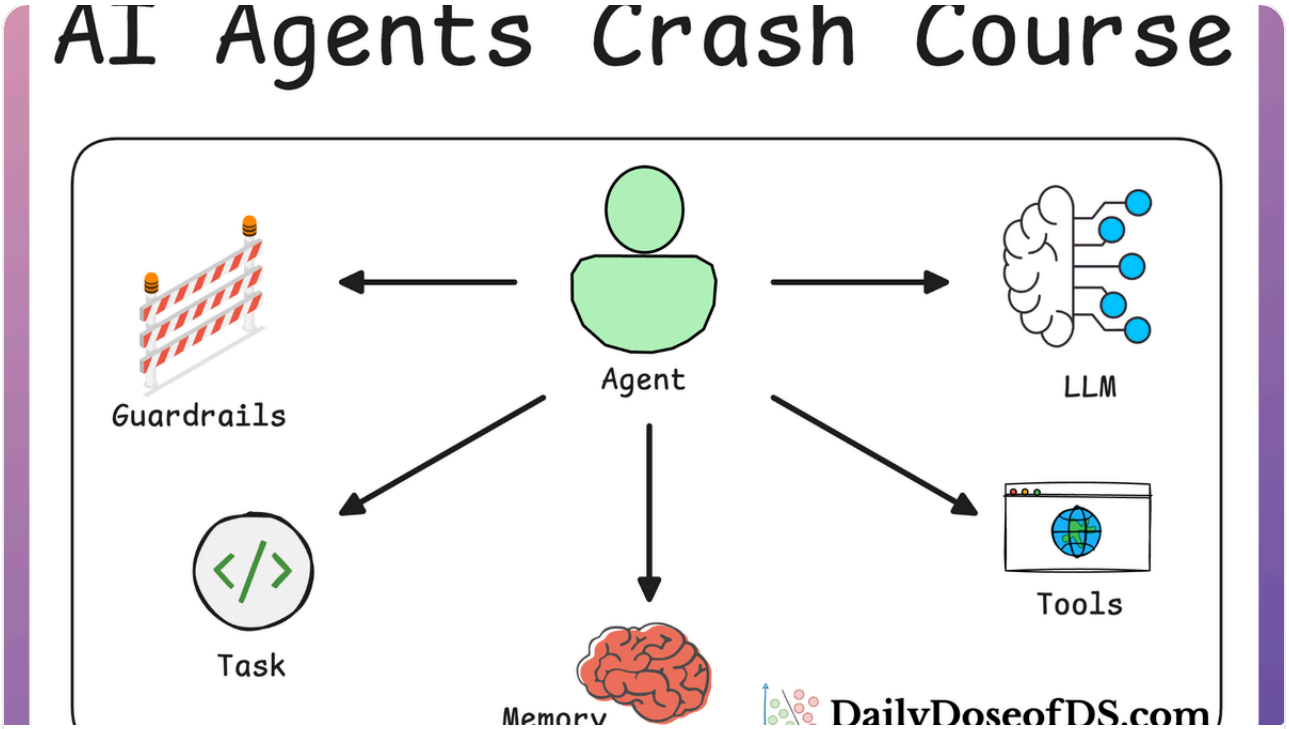
If you haven't read Part 1 and Part 2 yet, it is highly recommended to do so before moving ahead.



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)

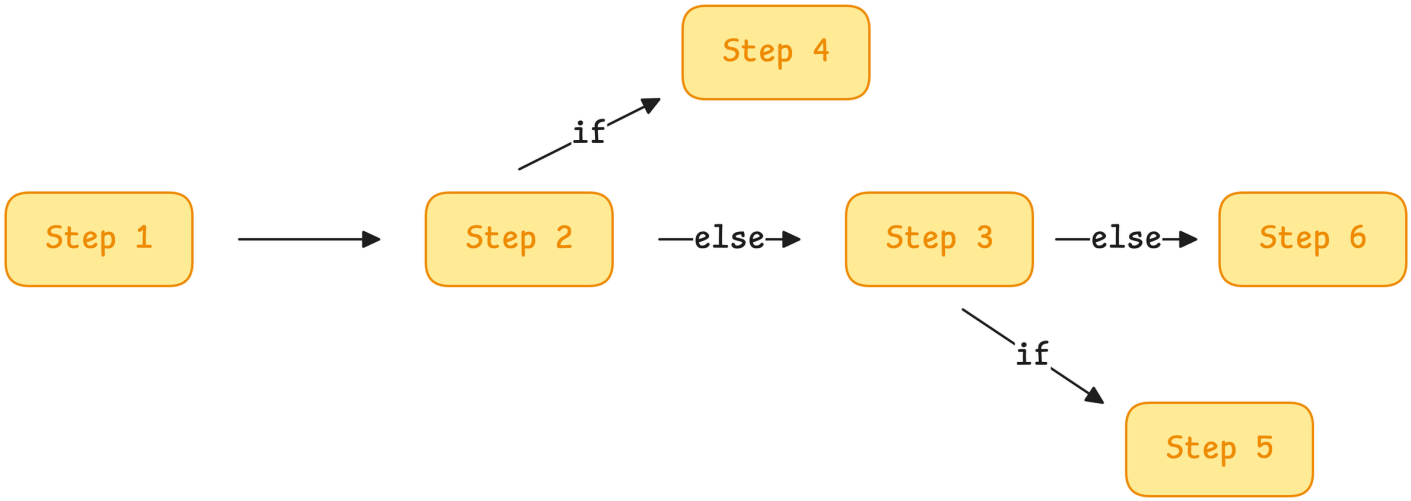
A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

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 behaviour 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!

Implementing Flows

Throughout this crash course, we shall be 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!

Setup

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

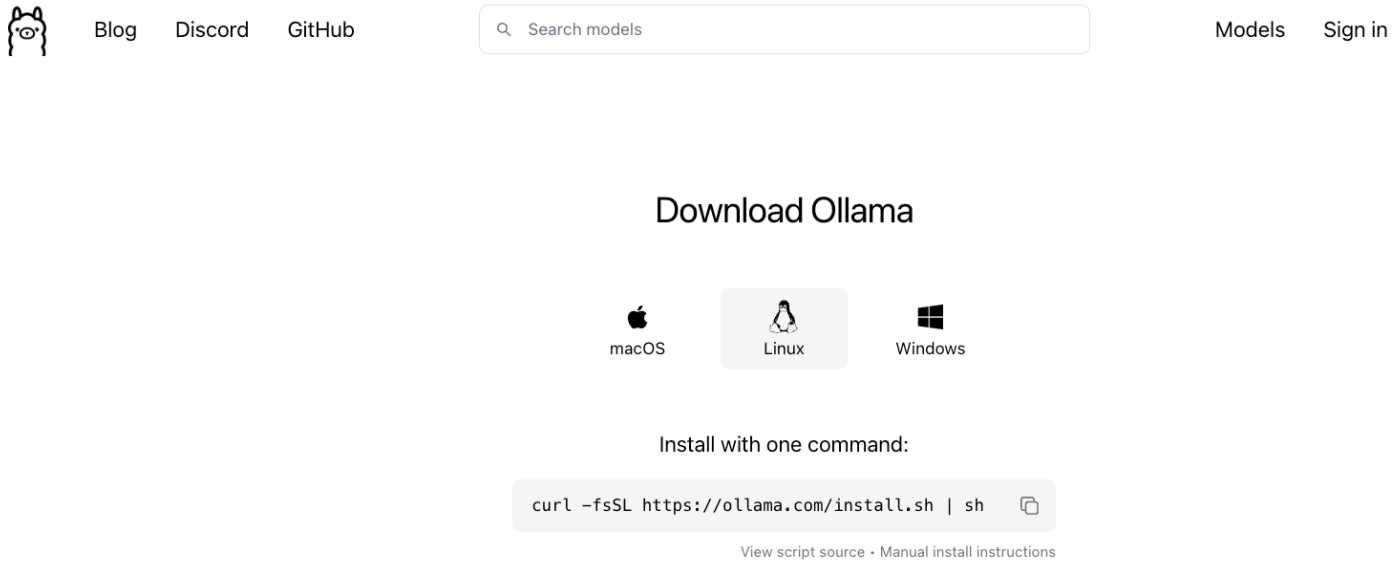
- and many more.

> 💡 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 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:

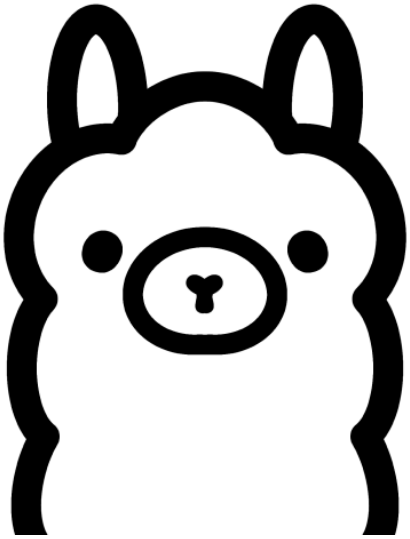
Also, here's a step-by-step guide on using Ollama:

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



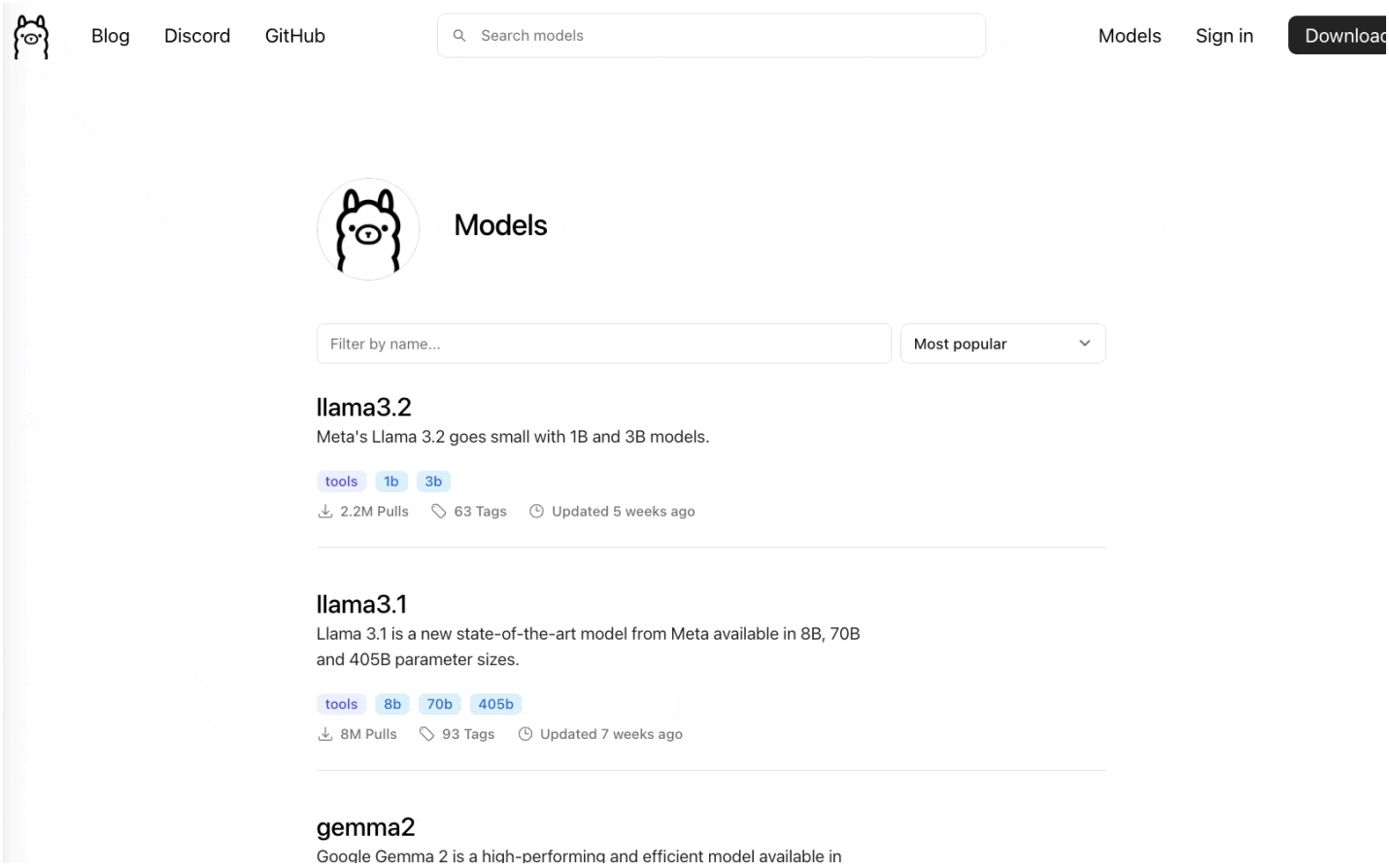
    - If you are using Linux, you can run the following command:

- 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.

You can download the code for this particular deep dive below:

AI Agents crash course Part 3

AI Agents crash course Part 3.ipynb • 16 KB

Basic Flow

In this walkthrough, we'll explore how CrewAI Flows allow you to effortlessly manage sequences of tasks powered by AI.

To keep things practical, we'll build a simple yet interesting scenario: first generating a random movie genre, then using that genre to suggest a popular movie recommendation.

If you can use OpenAI, we recommend doing that. If not, we have demonstrated everything with Ollama as well.

Get started by installing these dependencies and also make sure your `.env` file has your OpenAI API key configured:

We'll define a new Flow called `MovieRecommendationFlow`. It contains two methods:

- Generate a random movie genre.
- Recommend a popular movie based on the generated genre.

To do this, we start by defining a Flow named `MovieRecommendationFlow` that inherits from the CrewAI's `Flow` class.

In CrewAI, the method decorated with the `@start()` decorator indicates the entry point of the Flow. Any method with this decorator runs first when the flow is initiated.

Inside the `generate_genre` method, we send a request to OpenAI and ask it to provide a random movie genre.

If you are using Ollama, we can use this code instead:

We return the random genre from the `generate_genre` method.

Now let's move to the next part, which is the `recommend_movie` method. Here, we decorate it with the `@listen` decorator:

In the above code:

- The `@listen()` decorator indicates this task waits for the output of another task (`generate_genre`) before executing.
- It receives the output (`random_genre`) directly from the previous method.

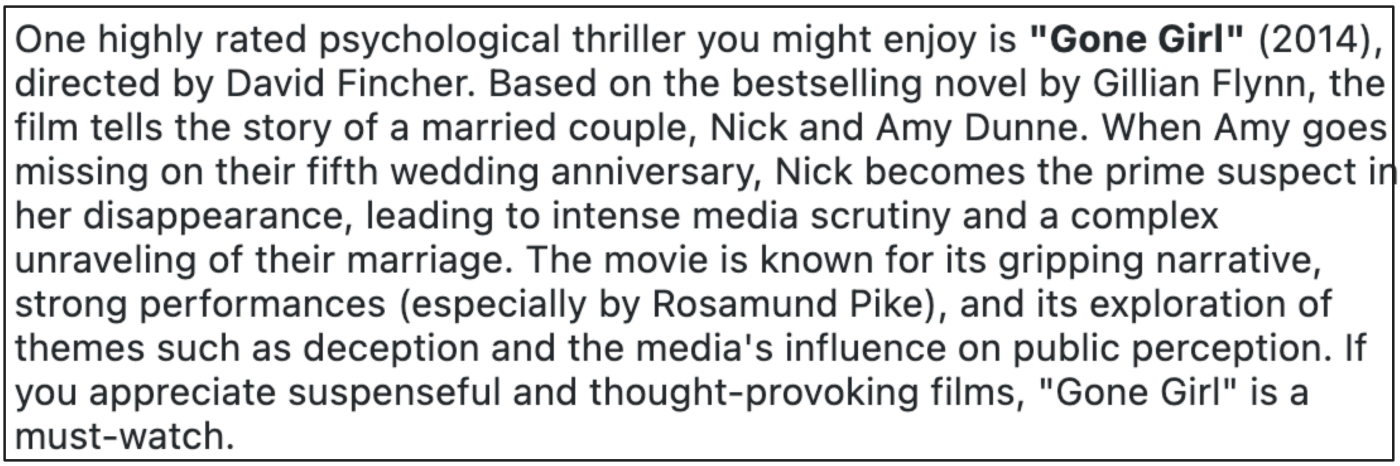
Inside the `recommend_movie` method, we pass the genre to the LLM to generate the movie recommendation and return the recommendation.

If you are using Ollama, you need to make the following changes like we did earlier:

Finally, we kick off the Flow as follows:

- `flow.kickoff()` starts the Flow execution, running the tasks in the correct sequence (first the genre generation, followed by the recommendation).

This produces the following output:



While this is a basic demo that could have been done with a simple LLM call, the example was used to explain that this particular design pattern is incredibly powerful for several reasons:

- Flows automate AI-driven sequential task execution, removing manual intervention.
- Tasks can easily share data, ensuring consistent context management.
- You can effortlessly add more tasks and dependencies.

Here's what you need to remember:

- `@start()`: Identifies the entry point(s) of your Flow. Multiple methods can have `@start()`, running concurrently.
- `@listen(task_name)`: Makes the decorated task wait for a specified task to complete, using its output directly.

Access intermediate states

In the above demo, we only accessed the final output of the model. But at times, we may want to access the intermediate outputs (or states). Moreover, at times, we may also want to update the intermediate states during Flow execution.

Let's look at that below.

Recall that we executed our Flow as follows:

Here, `flow` is our Flow object. This object has an attribute `state` that can be used to store any intermediate values:

```
flow.state
```

✓ 0.0s

```
{'id': '17d9ef32-3b86-4dc0-8917-3604e4e11edd'}
```

As shown above, currently, it holds a unique flow-id (which might be also needed at times).

The state attribute is a dictionary, and we can access this specific `state` attribute to store or update any of the existing values. Let's look at that below.

We have the same demo as earlier (with Ollama), but this time, we have stored the intermediate states in the `state` attribute:

- First, we have the `generate_genre` method, which is decorated with the `@start` decorator as discussed earlier. We store the randomly generated genre in the `state` dictionary with the `genre` key:

- Next, we have the `recommend_movie` method which is decorated with the `@listen` decorator as discussed earlier. We store the movie recommendation in the state dictionary with the `recommendation` key:

We run this flow as follows:

...and we get the following output:

```python
from IPython.display import Markdown
Markdown(final_result)
```

✓ 0.0s Python

Based on your request, I recommend the highly-rated movie "Hereditary" (2018) in the Science Fiction Horror genre.

Here's why:

- Directed by Ari Aster, known for his unsettling and atmospheric films like "Midsommar" and "Beau Is Afraid"
- Starring Toni Collette as Annie Graham, a family torn apart by grief and dark secrets
- Features a blend of psychological horror, sci-fi elements, and supernatural themes
- Received widespread critical acclaim, with an 82% approval rating on Rotten Tomatoes

In "Hereditary", the Graham family's seemingly idyllic life is shattered when Annie returns home from a mental institution, revealing a dark family history and forcing her children to confront their own personal demons. The film explores themes of grief, trauma, and the supernatural, making it a must-see for fans of Science Fiction Horror.

Give it a try!

Now, if we print the `state` attribute of the `flow` object, we get this output:

```
from pprint import pprint

pprint(flow.state)
```

✓ 0.0s

```
{'genre': 'The science fiction horror film.',
 'id': 'd77b9206-2f7c-4bef-8a66-1ff082055849',
 'recommendation': 'Based on your request, I recommend the highly-rated movie '
                   '"Hereditary" (2018) in the Science Fiction Horror genre.\n'
                   '\n'
                   "Here's why:\n"
                   '\n'
                   '* Directed by Ari Aster, known for his unsettling and '
                   'atmospheric films like "Midsommar" and "Beau Is Afraid"\n'
                   '* Starring Toni Collette as Annie Graham, a family torn '
                   'apart by grief and dark secrets\n'
                   '* Features a blend of psychological horror, sci-fi '
                   'elements, and supernatural themes\n'
                   '* Received widespread critical acclaim, with an 82% '
                   'approval rating on Rotten Tomatoes\n'
                   '\n'
                   'In "Hereditary", the Graham family\'s seemingly idyllic '
                   'life is shattered when Annie returns home from a mental '
                   'institution, revealing a dark family history and forcing '
                   'her children to confront their own personal demons. The '
                   'film explores themes of grief, trauma, and the '
                   'supernatural, making it a must-see for fans of Science '
                   'Fiction Horror.\n'
                   '\n'
                   'Give it a try!'}
```

As depicted above, the `state` attribute has both the genre and the recommendation which we stored earlier.

Let's learn more about the states below.

States in Flows

Managing the state effectively is essential for building robust and maintainable AI workflows. CrewAI Flows provides two approaches for handling state:

- Unstructured state management, which is more flexible and dynamic.
- Structured state management, which enforces schema validation using a model-based approach.

In this guide, we'll go beyond just storing intermediate values and explore how to modify, update, and manage the state effectively throughout a flow.

Instead of generating movie recommendations, this time we'll build a task management system where an AI agent:

1. Generates a task for a software engineer.
2. Updates the task's status to "In Progress."
3. Marks the task as "Completed."

Let's break it down!

Unstructured state

In unstructured state management, all state attributes are stored dynamically in `self.state`, similar to how a Python dictionary works. This allows the flow to store and modify state values on the fly, without requiring a predefined schema

Here's how we can implement a task management system using an unstructured state:

Here's a breakdown of the the above code:

- The `@start()` decorator marks `generate_task` as the first method to run in the flow.
- The flow automatically assigns a unique state ID (`self.state['id']`).
- The task details and its status ("Pending") are stored in the state dictionary.

Next, we update the task's status to "In Progress" (`start_task`):

- The `@listen(generate_task)` decorator means this function executes after `generate_task`.
- The state dictionary is updated, setting the `"status"` key to `"In Progress"`.

Finally, we mark the task as "Completed" (`complete_task`):

- This method listens to `start_task` and updates the task status to `"Completed"`.
- Finally, the entire state is printed to verify that all values were correctly updated.

We execute the Flow as follows:

...and get the following output:

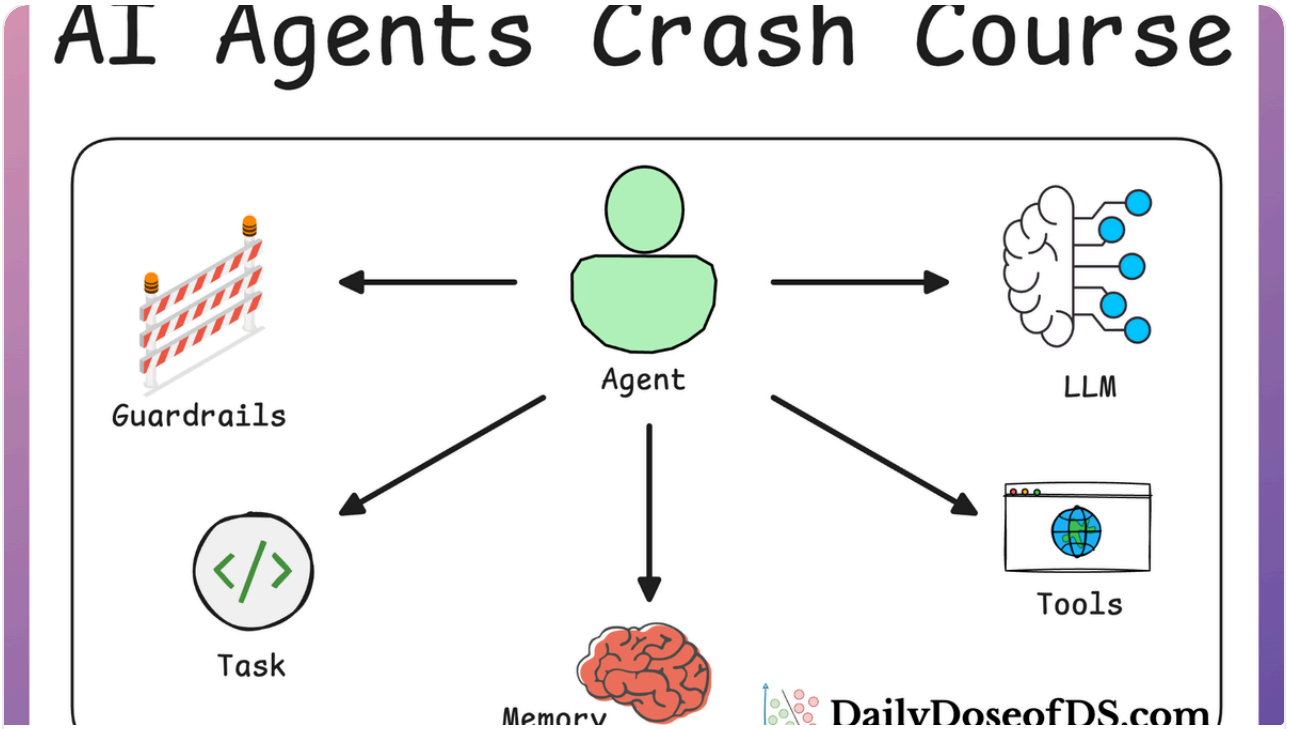
```
Flow started. State ID: 3b727952-fea4-4c87-84bd-8e8a6ddb6fa7

Task generated: Fix a critical bug in the payment system (Status: Pending)

Task status updated: In Progress

Task status updated: Completed

("Final Task State: {'id': '3b727952-fea4-4c87-84bd-8e8a6ddb6fa7', 'task': "
 "'Fix a critical bug in the payment system', 'status': 'Completed'}")
```

As we can see above, the Flow state has been dynamically updated throughout the flow using the unstructured state dictionary.

Structured state

While unstructured state management is flexible, structured state management enforces a predefined schema, ensuring that all state values conform to a specific format. This prevents accidental errors and makes debugging easier.

Like we learned in Part 2 at the time of structured outputs, we'll use Pydantic to define a strict schema for the task state.



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

Pydantic is a data validation library in Python that lets us define precise output schemas.

We start by defining the schema below while also specifying a type for all the state attributes:

Next, we have a similar code like we had earlier:

Before we execute it and look at the output, The notation `Flow[TaskState]` is a type parameterization (also known as generic type hinting) in Python. This means that `Flow` is a generic class, and `TaskState` is being passed as its type parameter.

`Flow[TaskState]` tells CrewAI that the flow's `self.state` attribute must conform to the `TaskState` schema.

We execute the Flow as follows:

...and get the following output:

```
Flow started. State ID: cda9e15c-4600-4538-ae5e-3ef20dfbe949

Task generated: Develop a new API endpoint (Status: Pending)

Task status updated: In Progress

Task status updated: Completed

("Final Task State: id='cda9e15c-4600-4538-ae5e-3ef20dfbe949' task='Develop a "
 "new API endpoint' status='Completed'")
```

As we can see above, the Flow state has been dynamically updated throughout the flow using the unstructured state dictionary.

Now the good thing about this particular Flow is that unlike the unstructured flow, we cannot dynamically add any new states during run-time.

For instance, consider this implementation below:

Now, if we execute this Flow as shown below:

...we get an error that there's no field `priority` in the schema:

```
File /opt/anaconda3/lib/python3.12/site-packages/pydantic/main.py:925, in BaseModel.__setattr__(self, name, value)
    922     self.__pydantic_validator__.validate_assignment(self, name, value)
    923 elif self.model_config.get('extra') != 'allow' and name not in self.__pydantic_fields__:
    924     # TODO - matching error
--> 925     raise ValueError(f'"{self.__class__.__name__}" object has no field "{name}"')
    926 elif self.model_config.get('extra') == 'allow' and name not in self.__pydantic_fields__:
    927     if self.model_extra and name in self.model_extra:

ValueError: "StateWithId" object has no field "priority"
```

This shows that:

- Unlike unstructured state, where any value can be added dynamically, here only predefined attributes (`task` , `status`) are allowed.
- If an undefined attribute is accessed (`self.state.priority` without defining it in `TaskState`), an error will occur.

Define unstructured states when:

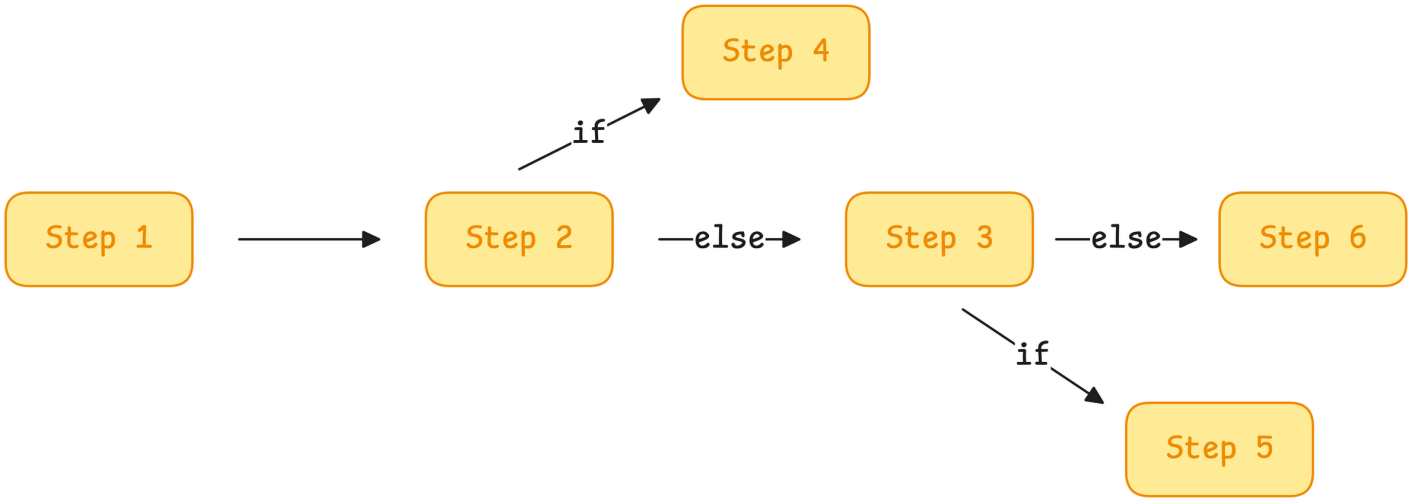
- The workflow's state is simple or highly dynamic.
- Flexibility is more important than strict definitions.
- You are rapidly prototyping and don't want to define schemas.

Define structured states when:

- You need strict validation of state attributes.
- Type safety is critical for avoiding runtime errors.

Conditional Flow Control

Managing the flow of tasks dynamically is a crucial part of AI-powered workflows. CrewAI Flows provides powerful tools to control execution paths based on conditions.



Below, we shall cover three important mechanisms for flow control:

- `or_` function: Triggers a task when at least one of multiple conditions is met.
- `and_` function: Triggers a task only when all specified conditions are met.
- `@router()` decorator: Directs the flow dynamically based on decision logic.

To avoid redundancy, we'll apply these concepts to a customer support system where AI determines how to handle and escalate user requests based on different criteria.

OR conditional logic

The `or_` function allows a method to be executed when any one of multiple tasks is completed. This is useful when a task should proceed regardless of which event happens first.

Let's build a system where a support request can come from two sources:

- Live chat
- Email ticket

If a request is received from either channel, it should be logged.

This is implemented below:

In the above code:

- Two independent sources (`live_chat_request` and `email_ticket_request`) can generate a request.
- `log_request` listens to either event using `or_()`.
- When a request arrives from either source, it is logged immediately.

If we execute the above Flow, we get the following output:

Depending on which method completes first, we will get the logged outputs.

AND conditional logic

The `and_` function ensures that a task only executes when multiple conditions are satisfied. For instance, we can escalate a support ticket only after user response and agent review.

In this example, a ticket should only be escalated if:

- The user confirms they still need help.
- A support agent has reviewed the request.

This is implemented below:

- The user confirms they need help (`user_confirms_issue`).
- A support agent reviews the ticket (`agent_reviews_ticket`).
- The escalation only happens when both conditions are met.

If we execute the above Flow, we get the following output:

If only one condition is met, escalation will not occur.

Router logic

The `@router()` decorator allows a method to decide the next task based on a condition. This is useful when different cases require different handling strategies. For instance, some logic (which could be AI-driven) can determine if a support request is urgent or non-urgent:

- Urgent tickets should be assigned to a live support agent.
- Non-urgent tickets should be sent to email support.

The key concepts required in this workflow will be:

- Pydantic for structured state management
- Dynamic ticket classification.
- Routing execution flow using `@router()`
- Listening to specific values using `@listen()`

Let's implement it below.

To begin, we define a structured state using `Pydantic` by creating a `TicketState` class.

The class has a single attribute, `priority`, which is initially set to `"low"`. This ensures that every instance of this flow will have a well-defined state with a priority value.

Also, as discussed earlier, using Pydantic for structured state management prevents unexpected attributes and ensures type safety.

Next, we define a `Flow` called `TicketRoutingFlow`, which inherits from `Flow[TicketState]`.

This means the flow's state will follow the schema of `TicketState`. Unlike an unstructured state (where you dynamically add keys), here, `priority` must always be a string.

Next, we have the `classify_ticket()` method with the `@start()` decorator which marks the entry point of the flow:

For now, the function assigns a random priority (`"high"` or `"low"`) to `self.state.priority` .

Next, we introduce a `@router()` for dynamic execution.

In the above code:

- The `@router(classify_ticket)` decorator means that `route_ticket()` runs immediately after `classify_ticket()` completes.
- The function returns a string:
  - `"urgent_support"` if the ticket is a high priority.
  - `"email_support"` if the ticket is a low priority.
- This return value determines which function will execute next.

Next, we define two listener methods that listen to catch the output of the router method.

In the above code:

- The `@listen("urgent_support")` and `@listen("email_support")` listen for specific values.
- The return value from `route_ticket()` determines which function runs:
  - If `"urgent_support"` → `assign_to_agent()` executes.
  - If `"email_support"` → `send_to_email_queue()` executes.
- To be more precise:
  - `@listen("urgent_support")` means this method will execute only when `"urgent_support"` is returned.
  - `@listen("email_support")` means this method will execute only when `"email_support"` is returned.
  - If neither value matches, nothing executes (though that's unlikely in this case).

Executing this flow, we get the following output:

It works as expected!

If we run it again, we may expect a different output since we have a random choice in the start method.

To summarize:

- The `or_()` logic triggers when → any one condition is met.
- The `and_()` logic triggers when → all conditions are met.
- The `router()` logic triggers when → we have decision-based routing.

Crews with Flows

The discussion we had so far did not involve AI-driven workflows.

Ideally, in real-world use cases when we need to build complex AI-driven workflows, managing different tasks with separate crews allows better modularity

and organization.

CrewAI Flows supports multiple crews, each handling specific responsibilities, and then orchestrating them in a single flow.

Below, let's learn how to:

- Generate a Flows project with multiple Crews.
- Understand the folder structure and where each Crew’s configurations are stored.
- Build custom crews by modifying existing templates.
- Connect multiple crews to create a seamless AI-powered workflow.

Thankfully, the CrewAI framework makes it seamless to integrate Crews into Flows.

To create a new Crew-enabled Flow in CrewAI, run the following command in the command line:

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 `test_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.
- `tools/`: This folder 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.

Let's look at the project created by CrewAI when we ran this command, and let's also execute the Flow.

This example demonstrates how to use CrewAI to create a poetry generation system with a well-structured flow.

Let's break down each component and understand how they work together.

We have the entire flow define in the `PoemFlow` class, which inherits from `Flow[PoemState]`. This class manages the entire poetry generation process through a series of well-defined steps:

The state keeps track of two pieces of information:

- `sentence_count`: The number of sentences our poem should have
- `poem`: The actual generated poem text

The Flow is organized into three sequential steps:

- Step 1) Generate Sentence Count

This initial step randomly decides how many sentences our poem will have (between 1 and 5).

- Step 2) Generate Poem

This step creates and executes the poem generation crew, passing along the desired sentence count. Also, notice the `listen` decorator on this method, which means it will executed after the `generate_sentence_count` has been executed. In this method, we also store the `poem` in the state.

- Step 3) Save Poem

Finally, the generated poem is saved to a file.

That said, for this to run well, you need to add the following three lines to the code and place your `OPENAI_API_KEY` in the `.env` file of the project:

Next, we have the `PoemCrew` which we learned to build in the Part 2. It's decorated with `@CrewBase` and defines the structure for our poem-writing operation:

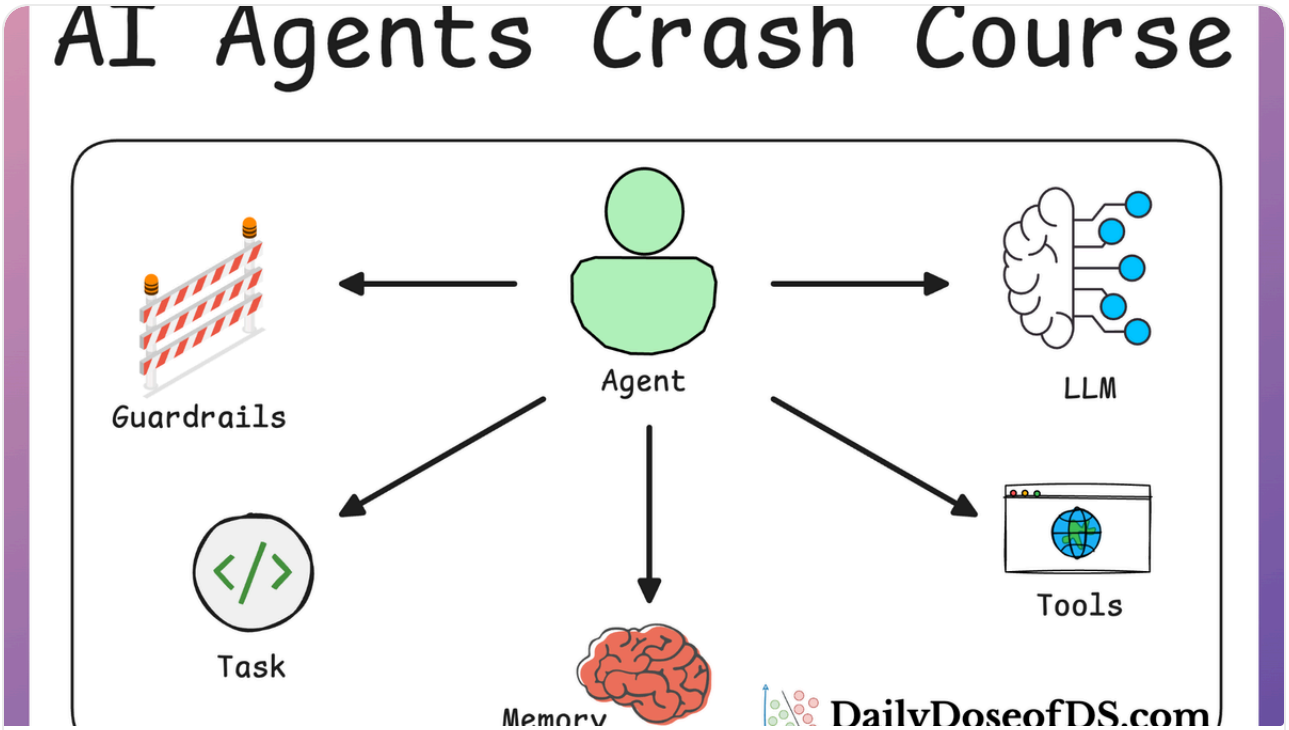
The crew consists of:

- A single agent (the poem writer).
- A single task (writing the poem).
- A sequential process flow.

Also, as discussed in earlier parts of this crash course, the project uses YAML configuration files for better maintainability where we define the Agent's role, goal and backstory, and task's description, expected output and the Agent associated with the task:

While there are no custom tools used here specifically, the project includes a framework for custom tools through the `MyCustomTool` class, demonstrating how the system can be extended with additional functionality.

We have already seen how to build custom tools in Part 2:



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

What's truly nice about this structure is that it makes it fairly easy to modify the poem generation parameters, add new agents, or extend the functionality through custom tools.

It's also a great example of how to structure a CrewAI project for maintainability and scalability.

Finally, to execute the Flow, move (`cd`) to the `src` directory of the project and run the base file:

This results in the following output in the terminal window:

```
(base) avichawla@Avis-MacBook-Pro src % python test_flow/main.py
 Flow started with ID: 4fff46ad-7d41-4b06-9cf3-880c36f578f2
Generating sentence count
Generating poem
# Agent: CrewAI Poem Writer
## Task: Write a poem about how CrewAI is awesome. Ensure the poem is engaging and adheres to the
specified sentence count of 4.



# Agent: CrewAI Poem Writer
## Final Answer:
In the world of code, CrewAI shines so bright,
With laughter and wisdom, it feels just right.
Crafting each query with flair and delight,
An awesome companion, our guiding light!


Poem generated In the world of code, CrewAI shines so bright,
With laughter and wisdom, it feels just right.
Crafting each query with flair and delight,
An awesome companion, our guiding light!
Saving poem
```

And we get a simple poem.

Alternatively, you can also move to the root directory of the project (which contains the README, src folder, etc.) and run this command:

💡 The above Flow execution will make use of the OpenAI's model. But its fairly simple to integrate Ollama as well here. Can you do that as an exercise? Here's what you need to do:

• Go to the `poem_crew.py` file.
• Define an LLM with Ollama (we discussed this in Part 1).
• Pass the llm as the `llm` parameter to the Agent.
• Done!

Let us know if you face any issues.

Conclusion, takeaways, and next steps

With that, we come to the end of Part 3 of the Agents crash course.

In this part, we explored CrewAI Flows, focusing on how to:

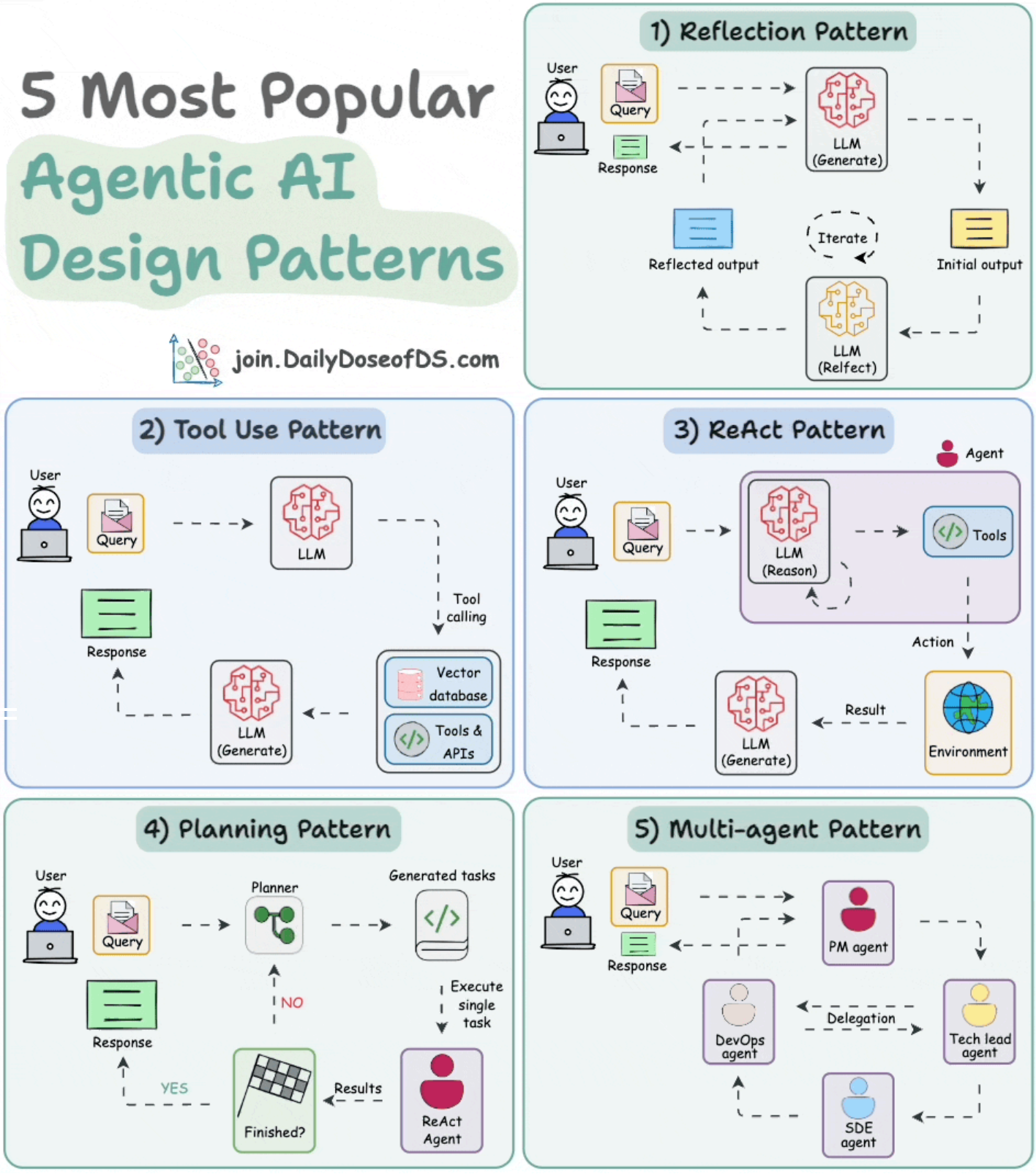
- Structure and manage flows using modular components like Crews, Agents, and Tasks.
- Handle state and data persistence across different steps in a Flow.
- Implement flow control mechanisms using:
  - `or_()` → Triggering a step when any of multiple tasks completes.
  - `and_()` → Executing a step only when all dependencies finish.
  - `@router()` → Dynamically routing execution based on conditions.
- Integrate Crews into a Flow, enabling complex multi-agent workflows.

Try implementing the exercise we mentioned in the last section section!

Even better would be to extend this to multi-crew Flows, which we shall cover in the next part where we shall cover several real-world use cases.

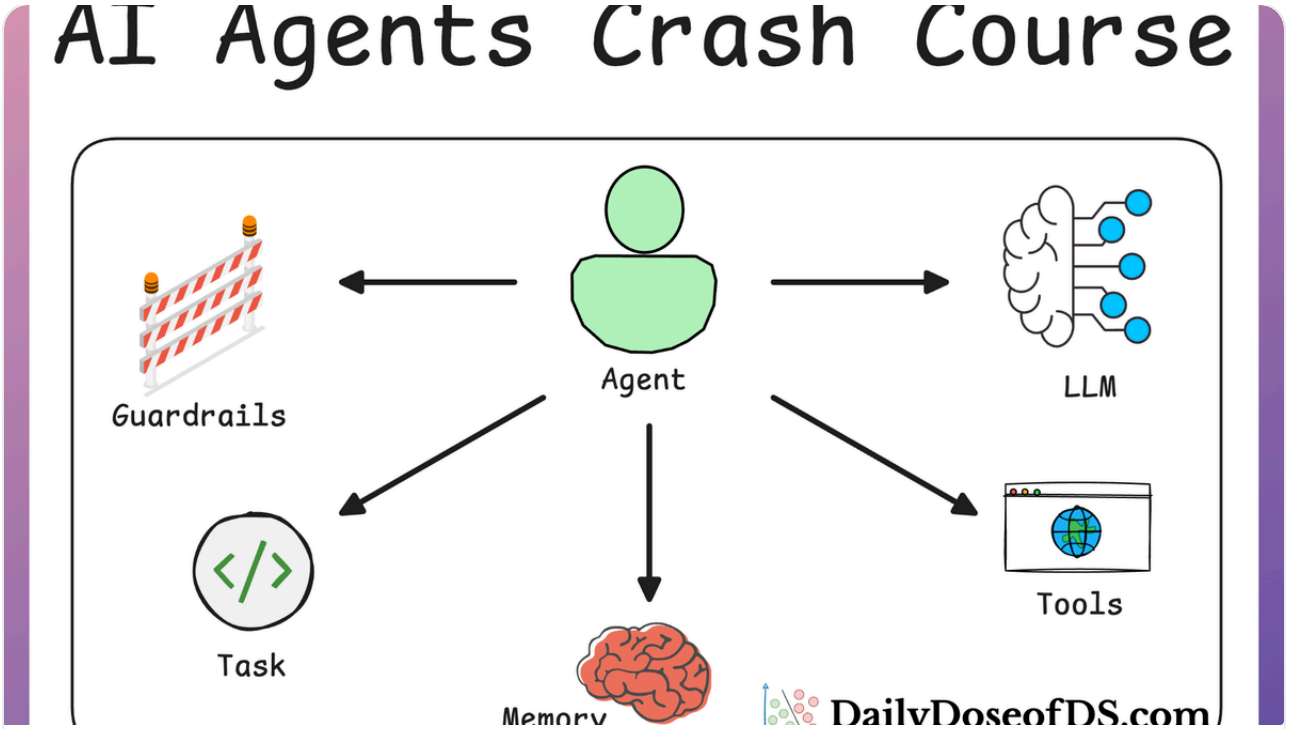
In the upcoming parts, we have several other advanced agentic things planned for you:

- 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 of this crash course here:



Building Flows in Agentic Systems (Part B)

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

Daily Dose of Data Science • Avi Chawla

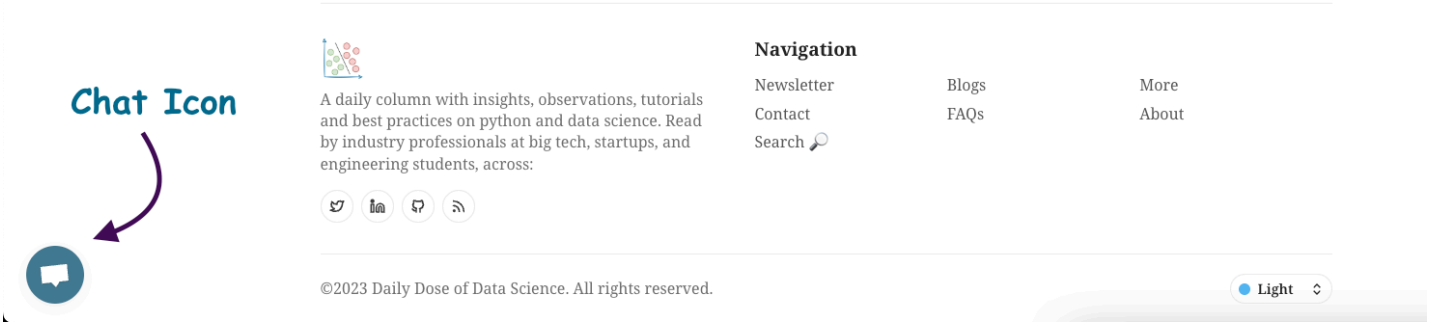
As always, thanks for reading!

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, Architecture, and Practical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar

Navigation